A Framework to Evaluate Candidate Agile Software Development Processes

A DISSERTATION

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Ian J. De Silva

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Doctor of Philosophy

Mats P. E. Heimdahl, Sanjai Rayadurgam

July, 2019

Acknowledgements

To my father, Joseph L. De Silva.

Thank you, first and foremost, to my family, church, and friends. Throughout this long process, you have been there supporting, encouraging, and praying for me. Thank you to Betty H. C. Cheng and Li Xiao who urged me on to graduate school in the first place as well as my lab mates, colleagues, students, and the department staff who constantly pushed me towards completion and provided much needed feedback and support. My gratitude and special thanks goes out to my mentors and advisors: Sanjai Rayadurgam, Mats P. E. Heimdahl, Dick Hedger, Stephen H. Kan, Neil Bitzenhofer, Maria Gini, and Michael W. Whalen. I would not be here without each of you.

Abstract

Today's software development projects must respond to fierce competition, a constantly changing marketplace, 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. Simulation provides a means to do this. Because of Agile's emphasis on the individual and interactions, we theorize that a high-fidelity model—one that models individual behavior—will produce more accurate outcome predictions than those that do not account for individual behavior. To this end, we define criteria, based on the Agile Manifesto, for determining if a simulation is suited to model Agile processes and use the criteria to assess existing simulations (and other evaluation frameworks).

Finding no suitable evaluation framework, we focus on constructing a simulation that satisfies our criteria. In this work, we propose a process simulation reference model that provides the constructs and relationships needed to capture the interactions among the individuals, product, process, and project in a holistic fashion. As a means for evaluating both our criteria and reference model, we constructed the Lean/Agile Process Simulation Environment (LAPSE), a multi-agent simulation framework for evaluating Agile processes prior to adoption within an organization.

The contributions of this work are threefold. Building on the simulation assessment criteria of Kellner, Madachy, and Raffo and the Agile Manifesto [56, 66], we establish criteria for assessing Agile simulations. From there, we define a reference model that captures the constructs and relationships needed to simulate Agile processes. We then show the satisfiability of our criteria and demonstrate how the constructs of the reference model fit together by building LAPSE. This work lays the groundwork for detailed *a priori* process evaluation and enables future research into process transformation.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

I once worked with an large, globally-distributed development organization that produces business software. Due to frequent marketplace changes and the desire to capture profits sooner, the organization needed to reduce the time between releases, so a process transformation group was assembled to propose and oversee changes. This group decided that Agile processes were best-suited to their needs. There are a number of processes and practices that exhibit Agility, so the team debated possible solutions, evaluated different case studies, studied the best practices defined by Agile experts, and even piloted potential processes on their own teams. Having found something that worked and was acceptable to management, they adopted the process throughout the development organization.

Guess what happened? The transformation failed to bring about the desired effects. Not only did the changes make developers less productive, but a few years into the changes the organization claimed to be in a "quality crisis" which delayed the product release.

What went wrong here? The organization looked at case studies, followed best practices prescribed by experts, and even piloted the changes, yet the process change failed to bring about the desired improvements. There are a number of things that led to the failure. First, there were important, yet unrealized differences between the case studies analyzed and the target environment. The case studies and process guides assumed process rigidity stemmed from culture, which was true here; however the organization also had to contend with other forms of rigidity such as tool-enforced processes. Second, a few influential team leads resisted the change and kept their teams on the old process delaying cross-team work integration and reducing

release frequency. Third, the team underestimated management's willingness to abandon existing process metrics. As a highly mature organization with decades of process and quality data, transitioning to a completely new process required adopting many incompatible metrics. For some, like expected defect detection rates over time, the incompatibility is not immediately apparent. As the project progressed and the defect levels rose higher than expected for project "phase", as is expected for this transition, management panicked and added back large chunks of the original process. This increased developer overhead, not only by prescribing activities globally that were inapplicable to all teams, but also by requiring more frequent and detailed progress reporting.

Had this situation been handled differently, we do not know if the outcome would have been any better. As a field, we do not yet fully understand what Agile techniques or activities are successful and why they are successful; little data and even fewer models are available to support investigating and evaluating these techniques.

Adopting an ill-fitting process can inhibit a team's ability to achieve desired outcomes (satisfy project goals) such as staying within budget, delivering on-schedule, or including necessary functionality. This makes Agile process adoption risky. How can we reduce the risk of adopting an Agile process that hinders the project or organization? Rather than arbitrarily selecting, tailoring[1], and adopting an Agile process, which could inhibit project success, we would like to reduce the likelihood of project failure from poor process fit by, *a priori*, evaluating candidate Agile processes, situated in their actual or modeled target environment, in terms of their expected project outcomes.

How, then, can we evaluate Agile processes prior to adoption? As we will discuss, only simulation is suited to our purpose as it alone can model project outcomes.

Software process simulations range from centralized, event-driven or continuous simulations to decentralized agent-based simulations. Given the degree to which individuals impact project outcomes [21] as well as Agile's emphasis on trusting the individual [66], we theorize that an Agile-process simulation that captures individual decision-making and the impacts of those decisions—a high-fidelity simulation that closely mirrors Agile development—will produce more accurate outcome predictions than those that do not explicitly model the individual.

In order to assess if high-fidelity, Agile simulations are more accurate than other simulations,

[1]Process tailoring is an approach for adapting a general software process specification (a process reference model) to a specific use [54].

we needed to classify existing work to determine which simulations, if any, can model individuals following an Agile process. To this end, we characterized the essential properties of Agile processes based on the most widely-used definition of Agile processes, the Agile manifesto [66], and constructed assessment criteria by which to evaluate simulation frameworks. Over the course of assessing different simulation frameworks/models, we found none that satisfied our criteria. This led us to the following questions:

1. Is it possible to create a simulation model that satisfies our Agile properties (are our criteria satisfiable)?
2. If it is possible to create such a simulation, what are the constructs (components) that might be needed in the simulation? Most importantly:
    a. How can we model lightweight processes (processes that do not provide complete prescription) such as Agile processes?
    b. How can we model the impact of each individual contributor on the project?

Determining if modeling individual decision-making improves prediction accuracy when simulating Agile processes is too large a task to complete within this work. Instead, we aim to lay the groundwork to make such a determination and, thus, to simulate Agile processes. Therefore, the primary contributions of this work are threefold:

1. We define criteria for determining if a simulation can model Agile processes;
2. We lay out the constructs that are needed to create a simulation, focusing particularly on individual decision-making; and
3. We show both that our criteria are satisfiable and how the constructs fit together to achieve this by building a reference simulation implementation, called the Lean/Agile Process Simulation Environment (LAPSE)[2].

The rest of this work is focused on addressing the questions enumerated earlier. Using the Agile manifesto within the context of the Kellner, Madachy, and Raffo (KMR) process simulation model selection guidance [56], we establish criteria (high-level requirements) for a process evaluation framework and use them to assess existing evaluation frameworks, including

[2]While Lean and Agile processes are similar (some even claim Lean is a form of Agile) and should both work within the simulation, evaluating if the simulation environment can truly model Lean processes is left to future work

simulation (Part I). Failing to find an existing Agile process evaluation framework, we set out to construct a simulation by first defining the base constructs and their relationships (a reference model) that establishes a high-level design (Part I). Using the reference model, we iteratively construct LAPSE, our multi-agent-based simulator (Part II). Finally, we set-up and execute a number of experiments that demonstrate how our reference simulator satisfies our criteria (Part III).

Part I

Process Model Assessment: Criteria and Existing Approaches

There are numerous tools/frameworks that could be used to assess processes. Some, such as guidance-based frameworks (e.g., Cockburn's selection framework [24]), use general best practices to guide process selection and tailoring while others, such as static analyses, check for process correctness according to some expressed properties. What should we choose? We need some selection criteria or requirements to help us determine what tools/frameworks are well-suited to assessing candidate Agile processes prior to adoption. To do *a priori* assessments, potential frameworks/tools will require some model or abstraction of reality over which to reason. Kellner, Madachy, and Raffo (KMR) enumerate questions to guide the selection of process simulation models, a specific type of process assessment model [56]:

1. What is the purpose of the model?
2. What is the model's scope?
3. What are the variables of interest (or model's output variables)?
4. What is the desired level of abstraction?
5. What are the input parameters?

Their approach can be generalized to support selecting any process assessment model. However, the specific input parameters are highly dependent on the type of analysis we will perform. Thus, in this part we focus on laying out the model's requirements (addressing KMR's questions 1-4) and use this to evaluate the different types of processes analyses. In a later part, we will return to the question of the specific input parameters required to assess processes.

Chapter 2

Assessment Model Requirements

2.1 Purpose

Our goal is to compare candidate Agile processes prior to adoption to support process selection and tailoring (for strategic management [56]). We expect that such an *a priori* assessment will reduce the risk (and associated costs) of adopting an ill-fitting process and encourage experimentation.

2.2 Model Scope

Implicit in our statement of purpose is the assumption that we want to assess a potential tailored, Agile process for a single project and team. While there may be value in assessing an organizationally shared processes—one in which a high-level process is given to and tailored by each team—this increases the complexity of the assessment as it must model both the tailoring of team-specific processes and the processes themselves. We want to focus on the assessment for a single team working on a single project before we incorporate additional features.

2.3 Variables of Interest (Assessment Dimensions)

Our stated objective is to evaluate candidate Agile processes to support process selection. We wish to perform this evaluation by comparing the expected outcomes of two or more candidate

processes. Thus, we do not need to forecast expected outcomes precisely, rather, we only need to show the relative benefits among modeled processes. This begs the question, what are the dimensions (or expected outcome metrics) over which we wish do perform this assessment?

According to the Project Management Institute (PMI), a project is "a temporary endeavor undertaken to create a unique product, service, or result" [2]. In other words, a project is a time-restricted effort taken to some end for some expected benefit. Thus, the project's primary concerns are its scope (what it is producing) and its value. Rather than directly measuring value, which can be subjective, projects focus on things that can adversely affect that value, primarily the project's cost and duration. Indeed, many process analyses (e.g., [16, 68, 56, 102, 58, 78, 84]) and practices use project cost, duration/schedule, and product (scope and quality) as their assessment dimensions. These dimensions are known collectively as the triple constraint or iron triangle. Due to their universal applicability and widespread acceptance, assessments must be able to compare candidate processes according to their the expected cost, schedule, scope, and quality at the end of the project.

It is unclear how we can estimate or compare these factors. What sort of metrics/measures do we need to quantify them? Let us go through each factor and discuss the types of metrics/measures suitable for comparison.

Cost In most software projects, the cost of developer and management time dwarf all other resource costs (including hardware). Thus, for software projects, the cost is the total effort (time) expended by the team in building the product.

Schedule Schedule is similarly straightforward; it is the calendar duration of the project from the time of the start of the project to its end. We must, therefore, be able to state unambiguously under what conditions the project both starts and ends in order to get an accurate duration. Both may vary by organization. One organization may, for example, end the project when all desired functionality is written, while another may only allow a fixed amount of time (like a fiscal quarter). It is up to the modeler to define the criteria for determining when the project starts and ends.

Scope Scope (or functionality) is notoriously hard to quantify, with attempts to do so ranging widely from source lines of code to point systems [5]. Some measures, such as source lines of

code (SLOC), are proxy quantifiers for scope and can vary depending on external factors like language, developer skill, and coding style [55]. This poses a challenge for assessing candidate processes, as the functional size can vary between assessments. For example, if you have two assessment runs, where one produced 24,000 SLOCs and another that produced 18,000 SLOCs, did the first one produce more functionality or were they just more verbose? Either could be true. Thus, we want a scope metric that is consistent across runs; that is, if two features have the exact same intrinsic properties, then they will have the same size measures regardless of which assessment run they appear in or who implements it.

We know that such metrics exist. Absolute point systems, such as function points ([48]), provide a means for consistently sizing features and could be suitable for process assessments.

Quality Traditionally, quality is defined as the degree to which the product is free from faults. This can be quantified as the number of escaped latent faults, which requires a constant scope between evaluations in order to be comparable (e.g., as described by˜Martin and Raffo [68]), or this can be normalized and expressed as the fault density, the ratio of the quantity of faults to the number of opportunities to inject faults (i.e., the number of units of functionality) [55].

Notice that we are concerned with the fault density rather than the defect density. Defects are detected faults, thus defect density only captures the ratio of detected faults to the units of functionality. With only the defect density, there is no way to distinguish between poor fault-detection practices and low-fault software. Insofar as we can measure the true number of faults in software, fault density provides a means of assessing quality. Further, by comparing the defect density and the fault density we can evaluate the process' fault-finding effectiveness, which may be of interest to some modelers.

It is worth noting that modern practitioners have expanded the definition of quality to incorporate technical debt—"a metaphor reflecting technical compromises that can yield short-term benefit but may hurt the long-term health of a software system" [63]. This can encompass things like architectural compromises, poor testing, and coding style violations [57]. While we do not intend to model the technical debt accrued from software aging and evolution, modeling the injection of poor-decisions as problems can aid us in expressing the detriment to quality within our density measure and provide a more useful measure to the modeler.

2.4 Abstraction Level

The assessment model's level of abstraction is dictated by the types of processes we need to model and the assessment dimensions. While we have addressed the latter, we must do the same for the former. In this section we will define Agile processes and use this definition to both infer properties of Agile processes and derive requirements for an Agile process assessment model.

Defining Agile As the mantra goes, "the only thing you can count on is change". This is no less true for software development. In contrast to traditional software development processes, which emphasize up-front planning, Agile processes—as lightweight, incremental software development processes—provide a means for addressing inevitable requirement changes in a structured manner. A software development process is Agile if it adheres to the values and principles enumerated in the Agile Manifesto (Figure 2.1 and Figure 2.2) [66].

Figure 2.1: The Agile Manifesto – The Values of Agile Processes [66]

...we have come to value:

V1. Individuals and interactions over processes and tools
V2. Working software over comprehensive documentation
V3. Customer collaboration over contract negotiation
V4. Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Deriving Assessment Model Requirements The key question here is: How can we translate the values and principles that define Agile processes into requirements for a process assessment framework? From the values and principles, we see a number of properties of Agile processes each impacting the process model (the set of activities and their sequencing constraints) and constraining the set of possible assessment frameworks (the means for evaluating a candidate process) (Table 2.1). While these Agile process properties are by no means canonical, we will

Figure 2.2: The Principles Behind the Agile Manifesto [66]

Undergirding the manifesto are twelve principles:

P1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

P2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

P3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

P4. Business people and developers must work together daily throughout the project.

P5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

P6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

P7. Working software is the primary measure of progress.

P8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

P9. Continuous attention to technical excellence and good design enhances agility.

P10. Simplicity–the art of maximizing the amount of work not done–is essential.

P11. The best architectures, requirements, and designs emerge from self-organizing teams.

P12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

explain how we arrived at each one using the values and principles enumerated in the Agile Manifesto.

Self-determination Far from the traditional view of interchangeable resources, Agile processes emphasize the effects of individual actions on project outcomes. Trusting the team (V1, P5, P11) empowers individual team members to define how they will complete the work and allows them to follow through with their plan. Self-determination results in lightweight, descriptive processes; processes that describe what needs to be done rather than how it must be done. Even with the additional freedom from (heavy) prescription, the team must work together to complete the product. Thus, Agile processes must provide a shared notion of a unit of work as well as

Table 2.1: Deriving assessment model requirements.

Property	Basis	Modeling Requirement
Self-determination	V1, P5, P11	Individual Behavior
Low-latency Collaboration	V1, V3, P4, P6	Interactions and Communication
Sustainable	P8	Impact on Individuals (and downstream effects)
Frequent Release & Feedback	V3, P1, P3, P4, P7	Product
Responsive to Change	V4, P2	Rework
High Product Quality	V2, P1, P9, P10	Product and Intrinsic Quality
Self-Optimization	P12	Adaptable Behavior

what it means to complete it (the completion criteria or definition of done); effectively, laying out an interface for coordination. Such an interface allows actors with heterogeneous processes, like business people and developers, to work together to produce a product (P4).

Because an individual's actions can significantly impact process effectiveness [23, 21] and because Agile processes encourage self-determination, potential assessment frameworks must allow us to model the actions of the individual and the impacts of those actions. Moreover, due to the lightweight nature of Agile processes, potential assessment frameworks must be able to handle processes that prescribe neither a complete set of development activities nor a total ordering of activities.

Low-Latency Collaboration Agile processes are highly collaborative (V1, V3, P4, P6), utilizing face-to-face meetings (and other low-latency communication) as well as regular feedback cycles to ensure that the product meets customer needs (P1). Just as with assessment frameworks used to evaluate traditional processes, potential process assessment frameworks must model interactions among the team.

Sustainable Development Out of respect for the team, organizations that adopt Agile processes must ensure that the pace of development is sustainable (P8). In order to understand the desire for sustainability, we should first look at unsustainable development. At the most extreme, a "death march" project, is "one whose 'project parameters' exceed the norm by at least 50 percent" [118][1]. The consequence of unsustainability is that the team is required to work harder or longer than they otherwise would (e.g., a 60-hour work week), which results in more

[1]Colloquially, developers characterize a sustained period of working significantly longer and harder as a death march.

injected defects, the forgoing of quality practices, or otherwise doing a poorer effort in detecting faults/problems.

In order to model (un)sustainability, we need to model the impacts of external constraints, such as a shortening of the project duration without scope change, as well as actor properties, like exhaustion, on the actor's choice of action, the action's outcomes, and its properties. The updating of an actor's properties based on its decisions provides a feedback loop within the model.

Frequent Release & Feedback and Responding to Change In most software projects, in-process requirements changes are inevitable, be they from client misunderstandings (or worse, clients changing their minds), markets demanding new or different features, regulatory authorities imposing new constraints, or dependency/environmental changes. Responding to such changes is one of the primary reasons for adopting an Agile process [1]. Agile processes build-in a means for both discovering and responding to changing requirements (V4, P2). Agile teams elicit feedback and discover changes both by working closely with clients (V3, P4) and through frequent releases of the working software (P1, P3). These frequent releases also provide a concrete measure of progress (P7), reduces waste (in part, by reducing the amount of time doing the wrong thing—P10), and allows clients to get an early return on investment (P1) [100, 82].

Releasing and eliciting feedback are specific actions we wish to model and are not fundamentally different from other actions we wish to model except that this feedback could result in changes to the planned work or the reworking of previously completed work. Because Agile teams expect project scope changes, we need to explicitly model both the work to be performed as well as rework within any candidate Agile-process assessment framework.

High Product Quality Emphasizing high-quality, working software (V2, P1, P9) enables the team to respond to change fast and satisfy clients. Rather than having to spend time enabling changes (paying down technical debt), the team can focus on making changes and getting feedback faster. Further, Agile's emphasis on simplicity (P10) both reduces the amount of wasted effort and reduces the number of opportunities for defect injection, improving product quality.

For process models, this quality-focus requires incorporating quality standards (and quality checks) into the completion criteria. Some teams will go further and prescribe specific practices

they wish followed in order to prevent/detect-and-fix defects in the software.

Frameworks for assessing these models must do more than model quality at detection time. Just as assessment frameworks must capture the effects of individual actions, so too must they capture the effects of upstream actions on downstream ones. This is especially apparent with product quality as, for example, defect prevention activities can reduce the number of faults that can be found by later quality assessments. Therefore, possible assessment frameworks must be able to capture the relationship among activities.

Self-Optimization From working with a number of different Agile teams, we have seen that the process the team starts with can be vastly different from the one they end with. In part, this work tries to address this by helping teams make informed process choices prior to starting the project. We hope this reduces the amount of process change. However, self-optimization impacts more than just the process. It may be small changes to the execution of those processes/practices—e.g., changing the format of requirements—or the completion criteria that result in improved outcomes. Agile processes include regular reflection over the team's processes and performance with the expectation of small, concrete improvements introduced over time.

While self-optimization is a key property of Agile, it is not a critical requirement of a process assessment framework. We assume that if another, potentially better, process is found, the modeler can rerun the assessment with the new process. Thus, at this time, we will not model process evolution (in-process change). In later work, we hope to revisit this assumption and model the effects of learning on process evolution.

From these properties, we require that a potential Agile process assessment framework model:

1. lightweight processes,
2. individual behaviors (especially in the presence of a lightweight process),
3. interactions and communication,
4. impacts of actions on individuals (and downstream effects),
5. the product,
6. intrinsic product quality, and
7. work and rework.

To model these properties, especially individual behaviors and their impacts, with respect to our desired outputs, we need a high-fidelity assessment model.

Chapter 3

Assessment Approaches

There are numerous approaches for *a priori* Agile process assessment. These forms of evaluation fall into a few groups: guidance-based frameworks, piloting, static analysis, and simulation.

Guidance-based frameworks Guidance-based frameworks provide direction for a human attempting to select and/or tailor a process. These frameworks collect best practices or observed outcomes into a model and recommends one methodology/practice over another based on some desired outcome. They may support methodology selection, such as Boehm-Turner-McConnell's Agile profiling [10, 72] and Cockburn's selection framework [24], or they may support practice selection within a methodology, such as Kalus and Kuhrmann's compilation of tailoring criteria [54].

Despite their ease of use and their basis in actual observations, guidance-based frameworks are too blunt of an instrument for our purposes as they do not support reasoning about the effects of combining/integrating practices, nor do they account for individual behaviors on project outcome. Combining practices could help or further hinder the team's ability to meet its objectives, while individuals can bring about success/failure despite the process used [7]. Thus, guidance-based frameworks do not satisfy our requirements for an Agile process assessment tool.

Piloting Another way to assess a candidate process is to try it on a small, representative subset of the team/organization that will eventually adopt the changes. In this case, the pilot team models the candidate process prior to widespread adoption. While this costs more than

other assessment approaches, it benefits from being situated within the target environment and provides a means for observing unexpected interactions among prescribed practices. Piloting, however, is not without its risks. Pilot programs do not demonstrate the process' ability to scale beyond the pilot team, increasing the risk of adopting the process throughout the organization. Further, if the pilot group is not representative of the entire organization, it can give you an incorrect result. Moreover, even if you could eliminate both the costs of piloting and the risk of selecting a non-representative team, sequential pilot runs can provide invalid data as runs are not independent. The discipline or skills learned from one processes can be translated into another and alter the results.

Piloting, is still a risky assessment approach. Can we do better?

Static Analysis Static analysis provides a means for analyzing an ordering of tasks (model states) to find process property violations, such as missing states, sequencing errors, and paths that lead to undesirable states [17]. However, static analyses are ill-suited to more complex analyses such as predicting process outcomes [85].

Simulation Simulation is an analysis technique that imitates "the operation of a real-world process or system over time" to allow the user to draw conclusions about real-world systems [8]. The complex, dynamic system to be analyzed is abstractly represented as a simulation model (the assessment model) [56]. With ever more complex simulation models, modelers must shift from manual to computational simulation.

But, is simulation the right tool for us? Simulation is appropriate when we want to study interactions within a complex system and their effects [8]. This complexity can take the form of model uncertainty, behavioral changes over time, and feedback mechanisms [85]. Because we want to analyze candidate Agile processes—where individual decision-making and team interactions are highly valued—simulation seems like a good fit.

There are, however, situations when simulation is contraindicated; when the model is ill-suited to answering the types of questions posed or can be done more efficiently through direct experimentation, is infeasible due to complexity or lack of data, is unverifiable, is cost-ineffective, or is insufficiently resourced [8]. We established its suitability and efficiency earlier. We will consider the remaining concerns as we assess existing simulations. Another potential downside to simulation is that the simulation is only as good as the input models, so care will be required

to ensure models reflect reality with sufficient detail to permit the relevant analysis.

3.1 Types of Simulation

Just as there are a number of potential assessment techniques, there are several forms of simulation. Simulations are classified according to the type of data it works over (qualitative vs quantitative), the amount of randomness in the model (stochastic vs deterministic), the model's distribution of control (centralized vs decentralized), and the model's representation of time (discrete or continuous time) [76, 101].

Simulations operate over qualitative or quantitative data. Qualitative data enables trend analyses and allow for the analysis of systems when there is insufficient data available [76]. In contrast, quantitative data allows the modeler to get expected-value predictions of the variables of interest.

With respect to randomness, models can be deterministic, stochastic, or a mix of both [77]. A model is deterministic if for the same input values, the output does not change between runs. In contrast, a model is stochastic if either the input values are selected from a random distribution or for the same input values, the output changes between runs (such as from uncertainty within the model).

Simulations also vary by the distribution of control [101]. In a centralized model there is a single "thread of control"; the simulator alone makes all decisions. In a decentralized model, different entities within the simulator have their own behaviors and goals. These entities act independent of one another, but coordinate with each other according to some prescribed protocol. Within centralized control models, we have observed models where activity enactment is located on the same thread as the controller and another where enactment is distributed over many threads (actors). To distinguish these forms, we will call the latter orchestrated control or orchestration.

Time is a key element to most simulations. It is no surprise then that there are differing views of time. Discrete time models advance time only when events occur while continuous models advance time at small, regular intervals, using functions to update model variables [67].

Let us look closer at types of simulations characterized according to their distribution of control and representation of time.

3.1.1 Centralized-Control Models

We begin our discussion with centralized simulations; simulations with a single "thread of control" [101]. Here, the simulator alone makes all necessary decisions.

There are three primary forms of centralized models, each distinguished by its representation of time: discrete-time, continuous-time, and hybrid approaches. In the rest of this section, we will evaluate each of these with respect to our assessment needs.

3.1.1.1 Discrete-Time

Discrete-time simulations advance time and update model variables only when events occur within the system. Because nothing of interest occurs between events, nothing need be simulated in this time [67]. One of the most recognizable forms of discrete-time simulations is discrete-event simulation (DES). Here, process models are expressed as procedural workflows (a network of activities) with entities, such as development artifacts, flowing through it [76]. DES supports models in which events are scheduled, transitions are monitored for satisfaction, or arrivals are placed on a network of queues for processing [6].

DES is one of several discrete-time, procedural models; models that describe a process in terms of control-flow either as a position within a fixed, finite set of states or as a set of transitions and their ordering [36]. More succinctly, these models describe how to achieve a goal. Just as with programming languages, there are also declarative discrete-time models. These models describe the logic or behaviors of the process (with relation to objects consumed or produced by actions) not as how to achieve a goal, but as what can be done and allow the simulation to determine how to achieve the goal [36]. Roughly, these models describe the data-flow of the process.

Rule-based models are an example of discrete-time, declarative models where the rules describe the set of possible process activities. When an activity is completed, the simulator selects the next activity from those whose preconditions are satisfied by the current state of the model [28]. Because they express process activities as rules and use them to progress the simulation, rule-based models are useful for understanding software engineering processes with fine-granularity and for exploring causal relationships of rarely used or "loosely structured" (lightweight) software processes [97]. Similar to Scacchi's rule-based model, the Multi-perspective Declarative Process Simulation (MuDePS) is a declarative simulator that constructs process execution traces from

a set of defined activities and sequencing constraints [3, 4]. In MuDePS, any valid activity sequence that is not explicitly prohibited by a sequencing constraint is permitted. Because it can compose any sequence of activities with little process prescription, MuDePS can model lightweight processes, but it does not model rework nor does it capture feedback loops (impacts of earlier behavior on later behavior).

On the whole, discrete-time models can handle both discrete and stochastic input, heterogeneous artifacts within the model, and interdependencies among activities (e.g., downstream effects of missing/insufficient resources) [67].

3.1.1.2 Continuous-Time Models

While discrete simulation models progresses time only when an event occurs, continuous simulation updates the model state by recomputing its constituent variables (levels) from model equations (flows) at equidistant time steps [76]. These equations are often integrals that account for continuous change since the last computation point.

System dynamics simulations, the most well-recognized continuous simulation, 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. Here, the process is captured as a set of flows—specifically, dynamic functions or consequential updates to variables dependent on time—that update levels and flow rates [67, 76]. Time within the simulation progresses in small, fixed-length intervals, and, on advancement, triggers the recomputation of flows and a new accumulation of levels.

Systems dynamics simulations have been used to simulate inspection processes [65], evaluate the impact of adding software inspections to a waterfall process [111], model a software evolution process [112], and provide decision support for balancing new-feature-development and support activities [61].

3.1.1.3 Hybrid approaches

While discrete event simulation and system dynamics both have their strengths and weaknesses, they excel in different ways. Some authors have attempted to combine these techniques, drawing on the strengths of each.

Martin and Raffo set out to model process workflows that contain feedback loops by combining

DES and systems dynamics [67]. To achieve this, they model each workflow as a sequence of activities similar to the models used by DES. Each activity is further modeled by a set of flow equations, specific to the activity, with predefined input parameters. When the simulation is run, it progresses in small time-intervals and recomputes the flow equations at the end of each interval. If an event occurs in the current interval, the events are processed, using the latest flow-levels, triggering any necessary transitions.

Donzelli and Iazeolla propose a similar technique, but, rather than integrating approaches at a single level, as Martin and Raffo propose, they carry out the simulation at two levels [32]. The top-level uses a discrete-event queuing network composed of activities, transitions, and artifacts, while the bottom-level models the details of each activity as a combination of an "analytical average-type function" and a continuous, time-dependent function.

While hybrid simulation has been used for a basic analysis of the utilization of manpower [67], a subsequent paper uses a hybrid simulation to evaluate process changes with respect to their impacts on project duration, effort, and quality [68]. They found, given their hypothetical data values, that the company being analyzed could indeed reduce duration and effort while increasing quality by both using only experienced developers and by eliminating unit testing. While this result is not generalizable, it does illustrate the strength of the hybrid simulation approach.

Each of the centralized-control models have been used to successfully analyze processes; however, few can model lightweight processes and individual behavior, and those that do fail to satisfy our other requirements (e.g., the ability to model rework). Thus, centralized-control models are ill-suited to our purpose.

3.1.2 Orchestration Models

The centralized models described thus far do all processing and enactment within the thread of control. However, there may be cases where it benefits the simulation to have centralized control but distribute the processing among different actors who can control the details of enactment. This enables the simulation to model unique properties of the actors which can be leveraged during enactment. For example, one actor could have a property that makes it more efficient at producing code, thus all coding activities assigned to it by the central orchestrator can be completed faster.

Noting the similarities between processes and software, the LASER group at the University

of Massachusetts, Amherst developed Little-JIL, a process programming language for specifying orchestration models for distributed execution [114]. Little-JIL captures process control flow representing it in a tree structure composed of process steps. Steps are composed of input parameters, pre/post-conditions, sub-steps (children nodes), and exceptional flows. With Little-JIL's well-defined semantics, a single model can be used for both process verification through static analysis tools like FLAVERS and process enactment through distributed-execution interpreters like Juliette [115, 15, 20].

Building on this foundation, JSim merges Little-JIL process specifications with an environment to execute a stochastic, distributed discrete event simulation [88]. In JSim, a central orchestrator uses the Little-JIL model to get the set of eligible next steps, sequences the steps, gets the required resources (including the actor/system that will execute the step) and parameters for each step, then schedules the steps by placing them in the actor's queue. Each actor then "completes" the assigned steps—determining both how much time the step will take and how to convert the input parameters to outputs—according to their behavior specification. Upon task completion, the orchestrator is notified, the set of eligible next steps is updated, and the cycle continues.

JSim's centralized orchestration means that actors do not determine their own behavior, and the upfront process prescription, while supporting parallel execution, is too rigid to support the lightweight nature of Agile processes. Indeed, the process designer would have to consider all possible combinations of the steps and exceptional flows in order to capture the freedom developers have under Agile to determine how to complete their work.

Moreover, because behavioral decisions are made by the central orchestrator, orchestration models more generally do not model individual behavioral decision-making thereby failing to capture self-determination.

3.1.3 Decentralized-Control Models

As opposed to a single thread of control over the simulation or orchestration, both of which cannot model the impact of lightweight processes on individual process decision-making, decentralized approaches devolve control to entities, called agents, that are responsible for their own behavior. These (autonomous, intelligent) agents act independent of one another, but must coordinate with each other either according to some prescribed communication protocol or by updating the world in which they are all situated. In process simulation, agents serve as proxies for humans

and automated systems, and, like humans within in an organization, have

1. limited observability and control over the world and/or the problem being solved
2. responsibilities delegated to them
3. decentralized data, and
4. asynchronous computation [51, 27].

As proxies, agents provide us with a means for simulating individual behavior and interactions.

Agent-based simulation (ABS) has been used to explain observed social phenomena [35, 34], analyze team changes [27], test emergency room procedure improvements to improve patient wait time [88], and simulate innovation strategies [39]. These approaches use a rule-based, discrete-time model within the agents to determine which actions to take. A similar technique, called method construction/assembly, has been used to build situationally relevant processes from process fragments [90]. These processes are fully-prescriptive, meaning they contain a complete sequence of activities within the process. Because we can compose processes from fragments, we can reasonably expect that agents within an ABS can construct processes or process-fragments from constituent activities thereby allowing us to simulate lightweight processes. Rather than constructing our own ABS, are there existing agent-based simulation frameworks that would satisfy our needs?

3.1.3.1 Existing Agent-based Simulators

Integrated Product Teams (IPT) To compare the impacts of team composition on the outcomes of an engineering design project, the Integrated Product Team (IPT) model defines both a multi-agent simulation model and simulator that incorporates the individual, team, product, and process concerns within a single model [102, 27]. IPT supports comparison through sensitivity analysis, rather than predicting project outcomes. The model defines three types of agents: those that complete tasks, those that assign tasks, and those that answer questions. Tasks—objects that combine process activities and product work packages—are prescribed at the outset and are arranged into a dependency network, where a task cannot be executed until its predecessors are complete. However, tasks may be executed in parallel, so agents select tasks to execute from their assigned set of tasks eligible to be run. This provides agents a limited ability to sequence their own work.

For each task, quality is estimated, but rework is not modeled either within a task or across tasks. This ignores an important reality of software projects; that finding and fixing defects is a key component of the process and early quality issues can have a significant impact on the overall cost [105]. As such, IPT does not satisfy our requirements for an assessment framework.

Virtual Design Teams (VDT) The Virtual Design Team (VDT) simulator [53] attempts to capture the emergent behavior and communication of design teams using a discrete-time multi-agent system. VDT 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 and does not affect any previously completed artifacts. In fact, VDT only measures the process quality, not the product quality [53], providing only indirect quality estimates. There have been many versions of VDT, and, like IPT, the first three embodiments (VDT-1,2,3) prescribe a static set of tasks upfront [62].

In the Virtual Team Alliance (VTA/VDT-3), Thomsen extended VDT to address goal incongruities [107]. While still adhering to the static, prescribed task structure, agents in the VTA can differ on how to complete the task resulting in potentially different task outcomes (e.g., reducing the volume of work that will be output for the task) [62, 107].

Expanding on the VTA's ability to model alternative ways to complete a task, Occam (VDT-4) models the informational overhead of delegating authority to less-skilled workers by explicitly modeling exceptional process flows [38]. It does this by probabilistically modeling when an exception (the change trigger) may occur, and, if triggered, updating the task network to address the exception. While it certainly attempts to solve an intriguing problem, it requires process modelers to define, upfront, all possible process flows, including exceptional flows. Rather than modeling the lightweight processes that stem from "trust[ing individuals] to get the job done" [66], the increased specificity of the model may cause the simulation to deviate from the actual processes followed by the team members. Worse still, an attempt to align the team's actual process with the modeled one could drive the imposition of more-prescriptive processes on the team. This runs contrary to both the Agile principles and our simulation model criteria.

TEAm Knowledge-based Structuring (TEAKS) People have a significant impact on project outcomes, thus, it is understandable that organizations are keenly interested in forming highly productive and effective project teams. TEAm Knowledge-based Structuring (TEAKS), another ABS, attempts to support team formation by predicting team performance—using fuzzy logic to reason over behavior-influencing characteristics (e.g., personality, emotions, and capabilities) and task properties (e.g., task difficulty) to determine task outcomes (team performance indicators) [69, 71, 70]. In this model, the task network is predefined and tasks are assigned to each agent to complete. TEAKS does not model the work product, in part or in whole, rather it models task outcomes as updates to team performance indicators—the task's goal achievement level, timeliness, quality, level of collaboration, and level of required supervision [70]. The team performance indicators are used both to influence the agent's internal state and, in the case of timeliness and quality, to assess the overall team performance. Because it does not model the work product and requires a complete task network at the outset, TEAKS does not satisfy our requirements for an Agile-process assessment framework.

The Articulator Project The Articulator project aims to create a reference model and simulation to evaluate processes and predict project outcomes [73]. This is achieved by specifying tasks—prescribed sequences of other tasks and actions (leaf node)—over which a central orchestrator will compose a shared plan [74]. The orchestrator then schedules resources and assigns tasks to agents to enact/execute the actions. During enactment, if resources are missing, are insufficient, or cause a failure, the agent(s) will invoke the Articulator (a separate orchestrator) to resolve the issue using a problem-solving heuristic. This will result in updates to the shared plan and or resource schedule (e.g., adding tasks to the plan or replacing resources).

Thus far, the Articulator sounds like an orchestrated simulation, with a centralized controller, yet it is unclear if this is the case based on the body of work on the topic. Two different knowledge-based simulation approaches are discussed—rule-based and (semi-autonomous) multi-agent-based. In their initial work, Mi and Scacchi state that agents choose which action to execute from those assigned to it and that the "choice can be influenced but not determined by outsiders" [73], while in their later work, the orchestrator and Articulator specify a plan which agents follow "according to the execution order specified in the plan" [74].

Regardless of the simulation approach, the Articulator project requires upfront process prescription in the form of tasks with little room for variation. This is apparent in the rule-based

approach, but it may not be clear in the semi-autonomous multi-agent approach. With the semi-autonomous multi-agent Articulator, agents can select which of the actions to perform among a set of tasks. This freedom to *schedule* or order a given set of tasks is quite different from the freedom to *select* which tasks to perform in that the former prescribes how each agent will complete the work while the latter, aligning with Agile principles, specifies what needs to be done and allows the agent to determine how to complete it.

Demonstration and Analysis Tool for Adaptive Systems Engineering Management (DATASEM) As Agile and Lean approaches have seen greater adoption and success with smaller teams, there have been efforts to scale them to larger teams and multi-team projects. For large-scale, software systems of systems (SoS), adopting these approaches is particularly challenging. A group of authors hypothesized the possible source of problems and developed a solution, based on Kanban, to address it [59]. Due to the number of complex interactions within their solution, validation on actual projects was infeasible, so they built a the Demonstration and Analysis Tool for Adaptive Systems Engineering Management (DATASEM)—a simulation environment for experimenting with different model interactions including the workflow (the process and product), organization, and environment [109].

DATASEM models work items within a task network that are serviced by teams which possess resources that may be required to complete the work [110]. Resource allocation and work scheduling are performed according to a governance model. The governance model also dictates updates to the work-item network to address rework and management/coordination.

DATASEM has two key components relevant to our discussion: the experiment/scenario builder and the Kanban-based scheduling system (KSS) simulator [110, 108]. The experiment generator takes the specified work-item network, organization, and governance models and creates a simulation scenario—the specific actor (agent) properties and work items as well as the expected evolution of the work-item network. The scenario together with parameters like resource allocation and prioritization algorithms are executed by the KSS simulator to estimate the completed scope, project duration, and the like. Because scenarios are fixed inputs to the KSS simulator, DATASEM deals with stochastic inputs by using a Monte Carlo simulation for statistical analysis.

With claims to simulate Agile and Lean processes at scale, DATASEM initially appears to be the right tool for what we want to do. However, DATASEM does not satisfy our requirements

for an Agile-process simulation framework as full-process specification is required. DATASEM's fixed work item network removes the "technical decision making from the simulation" [110] and, therefore, fails to capture the impact of individual decision-making on activity selection (self-determination) that could result in vastly different activity executions between agents.

Existing agent-based simulations fail to satisfy our requirements for an Agile-process assessment framework, yet agent-based simulation remains the most suitable assessment approach. Given that we want to compare Agile processes prior to adoption, can we construct a simulation framework that can model Agile as defined by our properties for an assessment framework?

Chapter 4

Simulation Inputs: A Foundation

Computational simulation consists of formally describing an abstraction of a real-world, dynamic system in a machine-readable way and symbolically executing that model to produce the desired outputs [8, 97]. Thus, we require both simulation inputs—including a simulation model describing the system we wish to analyze—and a compatible simulation engine (simulator) to execute the model. Because existing simulations fail to satisfy our Agile properties, we wish to build a simulator. Yet, we still have no notion of what the input parameters must be. To address this, we define a reference model—the constructs and the relationships among those constructs—that provide the basis of our vocabulary for the input parameters and simulation engine (Figure 4.1).

As both the type of agent and the workflow representation impact our final simulation model, we begin by analyzing each of these and selecting an agent and workflow representation that complies with our requirements (Section 4.1 and Section 4.2). Building on decisions of this chapter's earlier sections, we construct a reference model describing the constructs representing the people, product, process, and project concerns we wish to model as well as the relationships among them, and provide an example of our model's use within simulation (Section 4.3).

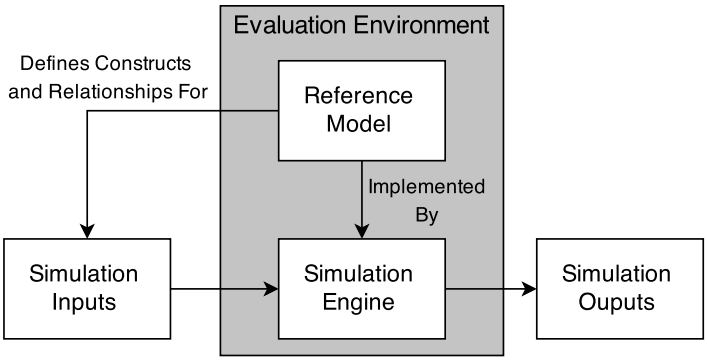

Figure 4.1: The reference model constructed here defines a vocabulary for simulation inputs and a high-level design for a simulation framework.

4.1 Types of Agents

We want to construct an agent-based simulation with multiple agents and little to no centralized control. More specifically, we want a multi-agent simulation composed of intelligent agents—agents that are proactive (exhibit goal-driven behavior), reactive (perceive their environment and respond), and social (interact to satisfy their objectives) [116]. This translates into a basic agent eval-loop:

1. Perceive/observe the world
2. Decide on next action
3. Execute action and update the world
4. Repeat

Agents differ primarily on how to do the second step. Both Russell and Norvig [96] and Wooldridge and Jennings [117, 116] classify agent architectures according to the agent's decision-making capabilities. Their architectural classes consider the following dimensions:

- World representation
- Reasoning approach (prescriptive or goal-oriented)
- Decision horizon
- Impact of historical actions on future decisions

They also include other considerations, like layering, that provide a means of combining different approaches but fall short of being dimensions in their own right.

To make sense of these categories, let us contrast two different architecture groups: deliberative and reactive. Deliberative agents have an "explicitly represented, symbolic model of the world... in which decisions (for example, about which actions to perform) are made via [logical] reasoning, based on pattern matching and symbolic manipulation" [117]. In contrast, reactive agents eschew symbolic world representations, often specifying behavior as a relation between a perception and an action to perform [116]. These relations may be specified, for example, as a set of transition triggers within a finite state machine.

As we have seen from our example paradigms, the world representation is closely tied to, and in fact enables, the agent's approach to reasoning. It provides the model over which reasoning occurs. Meanwhile, reasoning can be directed through complete prescription, toward (simple) goal-satisfaction, or toward the most desirable future state/world (the highest utility). With prescribed actions, agents select the next action using rules that prescribe specific reactions to events or based on eligibility (e.g., all preconditions are satisfied) without consideration of the expected destination/goal the actions support. For the latter two, agents select actions based on if they help lead to goal satisfaction or if they are expected to bring about the most desirable future state/world. With goal-oriented and utility-based reasoning (and with prescription to a lesser extent), agents deliberate over sequences of possible actions produced through a process called planning. The size of these sequences is determined by the decision/planning horizon.

In addition to planning, some agents may also learn from the outcomes of their past decisions to improve future action selection.

All of the agent-based simulations reviewed earlier use a model-based, reactive reasoning approach, modeling the agent's activities explicitly with a procedural process specification. Because of the properties of Agile, especially its lightweight nature, we want a simulation model where agents can reason (deliberate) over the current state of the world (including the state of the project and organization) to select an action expected to help them satisfy their goal. Moreover, as humans exhibit preferences that impact model outcomes and influence their action selections, we want to model task selection in terms of the desirability (utility) of the resultant state; we want a utility-based, deliberative agent.

4.2 Process Notations

Common among the analyzed agent-based simulations (Section 3.1.3.1) is a reliance on procedural workflows with little to no runtime flexibility. Indeed, these process models and the notations used to express them naturally lend themselves to model-based, reactive agent reasoning rather than deliberate approaches. Are there other process models, or more specifically process model notations, that are better-suited for deliberative reasoning?

Schonenberg, Mans, Russell, Mulyar, and Aalst classify notations along two dimensions: notation paradigm (specifically imperative vs declarative) and flexibility class [98]. Imperative (or procedural) models focus on the ordering of activities within the model—the process control-flow—prescribing how to complete the work rather than what needs to be done. In contrast, declarative models define a set of process activities and sequencing constraints over them. In these models, any process execution (sequence of activities) that does not violate a constraint is permitted. These constraints can be expressed over the activities themselves (e.g., DECLARE [80]) or over the data.

Observing patterns among notations, Schonenberg, Mans, Russell, Mulyar, and Aalst developed a taxonomy consisting of four classes of process model notation flexibility: flexibility by design, deviation, underspecification, and change [98]. Flexibility by design prescribes alternative activity sequences (flows) during model construction. At runtime, this may manifest as parallelism, interleaving, choice, etc. resulting in different executions over the same process. Flexibility by deviation permits activity reordering within the process model (e.g., skipping a step in the prescribed model or including optional constraints). Flexibility by underspecification allows the modeler to execute an incomplete process and rely on the execution environment to complete the process either statically or dynamically at runtime. Flexibility by change allows the execution environment to make changes to the original process model at runtime. These are longer-lived and larger changes than flexibility by deviation; for example, a change may be a wholesale replacement of a significant portion of the process.

With Agile processes we observe flexibility by underspecification from the lightweight process descriptions accompanied by flexibility by change stemming from self-optimization. Both forms

of flexibility are readily apparent within (vanilla) Scrum.[1] Within the iteration (sprint), as long as the developer meets the feature completion criteria, he is free to use whatever process he chooses (flexibility by underspecification). At the end of a sprint, the team holds a retrospective of the completed sprint and updates the process with expected improvements (flexibility by change). In contrast, existing process simulation model notations like LittleJIL are flexible by design not by underspecification or change.

With regards to Agile process simulation, we are more concerned with flexibility by underspecification than with flexibility by change because, for the most part, we do not need to model process evolution. We expect the modeler to update the simulation model and rerun simulation to observe the effects on the project outcomes. Moreover, this allows us to benefit from the updates throughout the project rather than only the portion after the change is made.

Both imperative and declarative approaches can be used for modeling processes that are flexible by underspecification, potentially using process placeholders to capture underspecification [98]. Prior to placeholder execution, the placeholder is replaced (or instantiated) with a process fragment. This fragment could be selected from a set of existing fragments or composed from process activities. Moreover, the placeholder may be replaced statically (once globally), dynamically (during each execution), or statically-by-agent (once per agent). With a declarative notation, we can take this to an extreme. We could treat the entire process as a placeholder and construct a process that produced the project's required output (goal state). Breaking this into multiple levels of goal states, each a placeholder for a fragment, gives us the notion of hierarchical process composition.

With an imperative notation, the placeholder approach requires a top-level process containing placeholders, but there is no guarantee that an Agile process will provide this. We could consider starting from some generic high-level process and attempting to instantiate placeholders that adhere to the process, but it is not clear that this meet-in-the-middle approach would be successful. Agile's lightweight nature lends itself toward declarative approaches, yet some practices within the process may prescribe a complete, imperative process fragment. Using a constraint-based notation—one where the process is defined as sequencing constraints over the

[1]Scrum is a popular Agile process that allows for a large amount of flexibility (Figure 4.4). 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); executes the plan, holding regular (scrum) meetings to briefly exchange status among peers; demo the product and gather customer feedback (the sprint review); and analyze the successes and failures of the previous iteration, making process changes if needed (the sprint retrospective)[92].

set of all possible activities—we can model both the underspecified and fully-specified aspects of the process. Moreover, these constraints can be considered during deliberation—composing sequences of activities that are expected to lead to a goal state (e.g., the feature completion criteria).

4.3 Reference Model[2]

Having established that we want to use deliberative agents and a declarative process/workflow notation for our simulation model, we can now define a reference model that captures the base constructs and relationships to be expressed in our simulation and describes the simulation inputs.

We wish to model the behaviors of heterogeneous individuals. Autonomous agents provide a means of both encapsulating the properties of the individual and modeling their effects on behavior and decision-making. We choose this representation because, like humans, agents are (1) bounded problem-solving entities, (2) situated in an environment with limited observability of the environment, (3) autonomous, and (4) reactive [51, 27]. Indeed, agents act as proxies for humans (or other actors) within a simulation. As such, they require a means of interacting with and updating the world. Agents do this by selecting and executing/enacting *actions*, and can, thereby, exhibit behaviors. Here, actions are reusable transforms, creating output artifacts based on the specific input artifacts. Thus, a *process* or, more precisely, a *process specification*, is a set of actions—we will call them *activities* when they are part of a process specification—and sequencing constraints over those actions that direct one or more actors towards a goal. In contrast to traditional project management which combine product and process concerns into work packages or tasks to be performed, the actions themselves only require that their input artifacts exist prior to execution and provide no other sequencing constraints. This will allow us to reuse the same set of actions with different process specifications.

So, actions act over artifacts. *Artifacts* are tangible objects/units that describe, embody, or support product function. These include software assets, "descriptions of partial solutions... or knowledge" used to to construct software [49], and deliverables, units of unique and verifiable product/result/capability required to complete a process/phase/project [2]. In incremental

[2]This section is an update on an earlier published work [29] that incorporates what we learned after implementing the reference model as part of a prototype simulator. The updates both simplify the reference model, eliminating unused constructs, and tie the model more closely to the Agile properties.

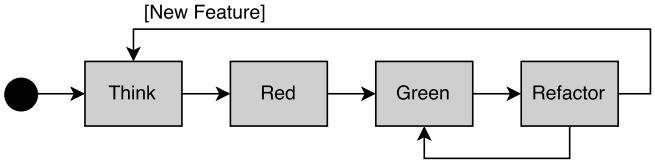

Figure 4.2: The test-driven development process (TDD) [100].

processes, such as Agile, artifacts are grouped according to the function or work they support. Further, artifacts may be related to one another, for example, through composition or dependency. Because we want actions to be reusable, we do not tie specific inputs to actions, which would merge the product and process concerns together; instead, actions match the attributes of the input artifacts (e.g., type) to determine if the action can run.

With multiple agents acting on their own, our model is inherently decentralized both in control and in enactment. This decentralization necessitates strong coordination among agents if they are to achieve a common goal (e.g., completion of the project). We model this coordination—agent interactions and communication—as messages passed between agents through a communication channel/medium. Explicitly representing a communication channel introduces additional model dimensions that may be of interest to the modeler such as the effects of interruptions, delays, and message prioritization on simulation outputs [53].

Are these constructs sufficient for modeling Agile processes? Not quite. While we have a basis for modeling behavior, further elaboration is required in order to address rework, coordination, and lightweight process specification—concerns included in our simulation model requirements (Chapter 2).

Individual Behavior and Rework We begin to look at how individual behavior and rework interact by examining test-driven development (TDD), a test-first development practice that is summed up as Think, Red, Green, Refactor (Figure 4.2) [100]. In TDD, the developer *thinks* through the function and selects a small increment to write. He develops a test case for the increment that fails on execution (*red*), then develops just enough of the code to ensure the test suite passes (*green*). Finally, he *refactors* the code ensuring all tests continue to pass. At the end of this process, the increment is complete and the process continues for the next increment.

Interestingly, within the Green-Refactor loop, we see an example of the work of one increment triggering rework in another, already-completed increment. We also see this with the other

practices like exploratory testing, where tests find problems that require changes (rework). Thus, we cannot simply contain rework within a single activity as VDT does [53]; we need to model it explicitly as either another artifact or as a new construct within the model. Further, explicitly modeling rework may allow other agents to assist in the rework. Because we want to evaluate processes with respect to the expected product quality, we will model the triggering of rework (especially from fault detection) as a new construct, which we will call *problem reports*. Moreover, as discovered rework/faults may not manifest as a program error (e.g., it could be a test artifact or technical debt), the modeler will need to decide what will be included in the resultant quality metric.

What about latent faults? How do we model faults that have not yet been identified? We can model latent faults implicitly or explicitly. Implicit models record the number of unfixed, injected faults throughout the process (e.g., Rus, Collofello, and Lakey[94]) and use that as the basis of quality predictions. However, such models do not distinguish between the different types of faults and would need to be translated to explicit problem reports in order to trigger the type of rework that we model. Explicit models do not have these issues. In an explicit model, latent faults are concrete objects injected when the faulty artifact is created. The agent is not made aware of the fault until the agent performs an activity that results in the fault's discovery. In addition to supplying the basis for defect density computations, explicit fault representations provide us with context on which activities need to be rerun to fix the bug and which other artifacts may be impacted. Further, an explicit model allows us to adopt a broader view of faults and expand out into other problems, like the introduction of technical debt, which may similarly trigger rework.

The TDD process also highlights the fact that activities produce different types of function. For example, the Red activity (test case implementation) in TDD generates automated unit tests for the product, but does not add function to the product itself. It would be incorrect to attach the test case implementation to a product artifact, since tests do not add product function and may undergo different process rigors than product artifacts. Instead, we wish to associate the test code with an existing test suite artifact if one exists, thereby grouping/integrating related artifacts (tests) under a single parent artifact (the test suite).

Coordination Software development is rarely an independent exercise. Often, producing software requires coordinating the work of many people. Processes help orchestrate the actions

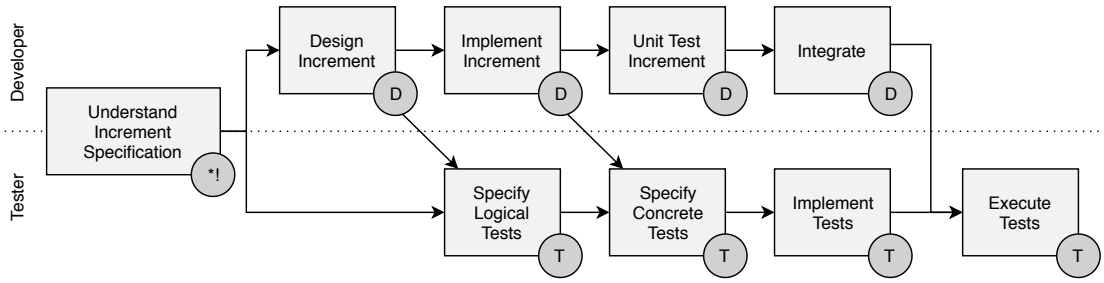

Figure 4.3: Example activity dependency network (arrows point to successors). The role annotations are shown for a tester (‘`T`’) and developer (‘`D`’). ‘`*!`’ indicates that the activity must be performed by every agent [100].

and interactions of the team. Of primary concern here is the division of work, as much of the coordination revolves around managing work dependencies and integration. Depending on the process, work can be distributed in a number of ways. It may be assigned from another actor or selected by individuals according to their skills or preferences, and it could be performed by one or more individuals (e.g., individual vs pairs). So, in addition to messaging (sending messages between actors), what do we need in order to model these different situations?

Assigning work to individuals, limiting work by specialty/skills, and intra-feature work division (i.e., one or more agents working to complete the same feature) all rely on a concept of a role. With centralized work assignment, there is a central orchestrator responsible for assigning work (a responsibility unique to the role). When an individual's work is limited to his specialty or a feature is divided among multiple people, the activities that can be performed by actors are limited to those permissible for the role(s) held by the actor. Figure 4.3 illustrates a division of development and testing in an activity dependency network. In this example, the tester is prevented from executing `Implement Increment` unless he is also in the developer role. It is worth noting that in some cases, such as our illustration, work is non-transferable; that is, the same actor must do all testing activities. If another actor takes over the testing activities, some amount of rework is required.

Contrasting the centralized orchestrator role with that of the division of a feature among many people (as in Figure 4.3), we see two types of roles: one that is permanently filled and another that is adopted temporarily according to the actor's current context. Let us further illustrate the latter. When teams first adopt Scrum, they may rotate the person filling the Scrum master role between sprints to ensure everyone understands the role [26]. In this example, the

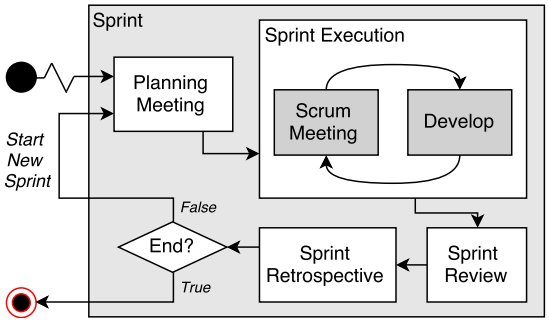

Figure 4.4: The Scrum process [99]. Initial set-up has been omitted.

role shifts based on some predefined criteria, mutual consent, or instruction from management between different contexts.

The team is composed of people in three roles that work together to produce the product [99]:

- Product owner – the customer representative responsible for describing and prioritizing customer needs
- Scrum master – the "servant-leader" responsible for facilitating useful interactions and coaching the team
- Developers – the professionals who produce value (produce the product according to the customer's needs)

In describing role changes and mapping out the Scrum process (Figure 4.4), we observe that developers work within a hierarchy of contexts (one context may be composed of other, child contexts), where sibling contexts, like activities, are partially ordered. Unlike activities, contexts define which roles are available (e.g., the product owner, Scrum master, and developer in Scrum) and, thereby, restrict the activities the actor may perform within the context. Context entry may also be restricted by the actor's role. Building on our Scrum example, we see that during the sprint execution, the Scrum master coaches the team and helps remove barriers to team progress while the developers work to complete the work scheduled for the sprint. However, when they enter the Scrum meeting context, the developers answer the three status questions ("What did I do?", "What will I do?", and "What is blocking my progress?") while the Scrum master facilitates the meeting and schedules follow-up conversations [99].

Rather than relying on a centralized orchestrator, Scrum, like many Agile processes, allows developers to select the work they will perform. This requires additional coordination structures

to reduce the duplication of work; often manifesting as repositories (e.g., backlogs, Kanban/Scrum-boards, and defect tracking systems) containing descriptors of the work to be performed at a given stage and, if claimed, the person responsible for completing the work. Developers claim work by following pre-established guidelines captured in the team's process. The coordination needed to establish work claims can be achieved by communicating either with a central agent (the repository) that tracks both work and work claims or with all agents on the team and reaching consensus. To support decentralization, the reference model requires only some means of expressing a claim or advisory lock.

Resources, Knowledge, and Lightweight Processes In our discussion of work assignments, we only restricted decisions according to the role(s) the actor held and the sequencing constraints of the process specification. However, work assignments may also be restricted by resource constraints and developer qualifications [92].

In project management, *resources* are any person, tool, environment, equipment, or supplies required to complete the project [2]. In our model, human resources and some computer systems (e.g., the centralized work repository in the previous section) are modeled explicitly as agents. Other resources, such as a build server, may still be needed in order to execute an activity, and, like agents, there may be a finite set of these resources that can be shared among the actors. By integrating a locking mechanism into our reference model, we can represent non-agent resources as artifacts which can be acquired prior to executing an activity.

When is someone qualified to work on a task (an action performed on a specific set of input artifacts)? For software developers and other intellectual workers, an individual is qualified based on his quality of knowledge across several knowledge domains. Procedural knowledge (e.g., how to test), declarative knowledge—including product knowledge, and domain knowledge for both the product and technical domains—are necessary for completing an activity, while organizational knowledge, such as the structure of the organization and inter-agent relationships, is needed for communication [89, 43]. Lacking a means to represent this knowledge, the reference model requires additional constructs to correct this. However, we cannot simply add knowledge requirements to artifacts and activities, as doing so increases coupling between the product and process, negatively impacting our ability to express lightweight, individually-tailored processes. If, for example, we add required skills knowledge to the artifact, then we must specify all required skills for any possible activity that could be performed on the artifact. Similarly, if

we add required 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 input artifact(s) and the activity. One way to resolve this is to attach domain and application knowledge (including knowledge of technologies and libraries) to the input artifacts and procedural knowledge (e.g. architecture, programming skills, or testing techniques) to the activity. The agent can then check if it has the knowledge/skills required to perform the activity on the given artifacts.

An agent simply possessing the right knowledge permits executing an activity on an artifact, but this is an overly simplistic view of knowledge. What if the agent is not skilled enough with a specific language or tool? This could either prevent the agent from completing work or it could be a source of faults (or inefficiencies) in the resultant artifacts. Thus, we care about both the possession of knowledge and the quality of that knowledge (or competency-level). Indeed, the IPT model captures this by requiring that the agent's competency-level match or exceed the required skill-level [27], and it is similarly useful to include in our reference model; however, the process for knowledge acquisition and use by an agent will be left to the specific implementation of the simulation framework.

To this point, we have discussed specific knowledge as the basis for assigning/selecting work. In the author's experience, some teams also assign work based on the developer's ability to learn the required skill/competency. There are a variety of ways to model this using the existing reference model—including modeling an agent's learning ability as knowledge required by some skill-learning action—so we will leave the reference model as-is and revisit the question of making additional changes if the need arises.

In developing the reference model, we have explored the constructs and relationships necessary to model Agile processes (illustrated in Figure 4.5). Based on our properties, we believe the model presented here captures the constructs and relationships. It represents individual, lightweight-process-compliant behavior with agents that select actions (construct a process execution) according to the agent's preferences, the natural sequencing constraints of the action, and the process. Upon execution, the action's impacts result in both new artifacts and updates to the agent's inherent properties (individual impacts). Further, the agent can improve or learn through sharing knowledge, which, together with collaboration, is achieved through inter-agent messaging.

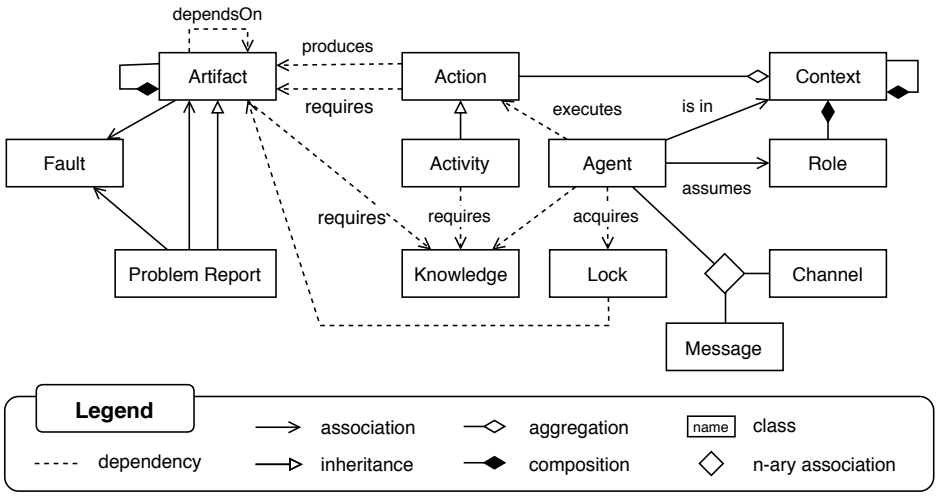

Figure 4.5: Reference model constructs and relationships (depicted as a UML class diagram). Not pictured: the increments and related goals the agent is trying to achieve by executing actions.

Artifacts provide us with an explicit model of the product and the objects that contribute to it. Further, artifacts provide the subject of desired rework (the reworkable object). In our reference model, we express rework triggers as concrete `Problem Report` instances.

4.3.1 Reference Model: An Example

Let's walk through how a simulation based on this reference model might work. To facilitate this discussion, we will first introduce an example scenario, based on the author's experience, and discuss how this scenario would progress once simulated.

Scenario 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, and product's requirement dependency network is partially depicted in Figure 4.6.

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 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" (shared

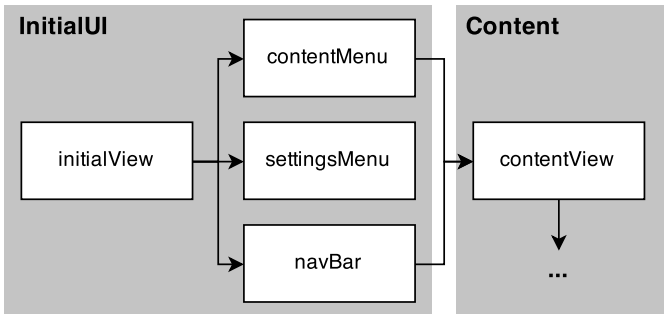

Figure 4.6: The abbreviated requirement dependency network where increments (small boxes) are grouped according to the story/requirement (large boxes) they belong to. Here, arrows point to successors.

completion criteria). With this team, the experienced developer with strong testing skills serves as a tester, another follows TDD, and the remaining developers determine their own activity sequencing, using the tester to verify their code.

Walk-through Now, let us 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 outer, `Scrum` context (Figure 4.4): the four developers 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 (update the product backlog agent)—and the `Sprint Planning Meeting` is held—populating the sprint backlog (the sprint backlog agent). On completion, the team enters the `Develop` child-context within `Sprint Execution`. At scheduled times, the agents will suspend existing actions and enter the `Scrum Meeting` context.

Within the `Develop` context, since there is only one artifact that is not blocked by prerequisites and is, therefore, eligible to be worked on (`initialView`; Figure 4.6), two agents work on it: the tester and an inexperienced developer. Each of these agents work on the activities defined for their role (as in Figure 4.3), including both of them attempting to understand the specification. Upon completion of the activity, the tester 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.

Assume the tester notices that all of the prerequisites have been satisfied for the `Implement Tests` activity, so it executes the activity with the `initialView` requirement artifact. As part of the simulation, the tester determines its outputs for the current tick based upon its knowledge levels compared to the required knowledge levels (the required knowledge of the artifact(s) and the activity), the activity's expected postconditions, the input artifacts' properties, and the 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 given inputs), it could be one or more artifacts/problem reports (e.g., a test suite containing the newly implemented test or problem reports), or it could be messages sent to other agents to improve knowledge. Assume here that generated problem reports are added to the sprint backlog so other agents may work on them.

While working on the project, each agent tracks the amount of time it spends on each activity, in terms of ticks, meanwhile gaining development skill and domain knowledge which would impact later agent performance. The process dictates both how to break down requirements and similarly when to award credit for work completion. For Scrum, the latter is done either during the Scrum meeting or upon exiting the sprint context.

Because no other work is available and all roles for the available work have been assigned, the other developer agents idle. This time is not counted toward their time on the project (their effort towards 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 preferences (e.g., their personal processes).

In order to simulate requirements changes, we may wish to include an agent that generates changes (including requirement additions and removals) according to some specified criteria and arrival rate. Change requests, modeled as a type of communication, may include information about a change to existing work (a logical set of artifacts, e.g., increments), changes to the dependency structure, and/or new work, 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 on the existing work in the product backlog if they have not been moved to the sprint backlog, otherwise, it schedules rework—by adding a problem report to the backlog—to change/remove completed or

in-progress 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 in-flight process changes 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, product functionality (scope), and defect density (quality) from the properties of the agents and the resultant product.

Duration and effort are straight-forward to compute. As mentioned, our functionality metric increases as the amount of completed work increases and similarly decreases with work removal. For scope changes, we can consider the work as incomplete until the associated updates to the product's artifacts are complete. Leveraging this functionality metric, we can compute defect density by summing up the number of unfixed faults/incomplete rework—those faults/rework items reported and latent faults still associated with artifact functions at the end of the simulation—and dividing it by the scope.

4.3.2 Related Work

In this section, we constructed a reference model to base our simulation engine on. However, software process modeling languages (SPMLs) abound. Why should we introduce a new reference model instead of adapting an existing SPML?

Many SPMLs express processes as control-flows. Control-flow based models or traditional workflows, like those expressible in BPMN [13] or UML (e.g., SPEM [103]), express how to achieve a desired outcome through process prescription. While prescription may indeed achieve the desired outcomes, it does not allow us to model unexpected ways in which the goal is achieved—ways in which the modeler did not foresee. In an Agile process simulation, capturing what should be achieved without prescribing how to do it allows us to model both lightweight processes and trusting the individual to get the job done (principle 5 from the Manifesto [66]).

Rather than strictly modeling control flow, case management systems model processes as

partially ordered sets of activities (cases) that consume and produce data [81, 75]. Here, actors select cases according to the state of the world (data objects) and their preferences then enact the activities that make up the selected cases. While this gives case management systems greater flexibility than traditional workflows, they still use imperative workflow models within the case [80]. Further, case management systems provide operations for process deviations (flexibility by deviation), but they cannot capture the unforeseeable ways in which an actor might reach the goal state (flexibility by underspecification).

Constraint-based, declarative process models, such as DECLARE's model [80], sequence activities using rules expressed over the activities themselves. These models often use temporal logic to express these constraints. Because they are declarative, they inherently model flexibility by underspecification. However, they do not model the data shared between activities and the constraints stemming from the sharing of data. This requires manually accounting for all data dependencies and their impact on activity sequencing as well as tracking the activity outputs [75]. Our reference model focuses on the relationships among the different entities to enable us to model the data constraints between activities. We expect that a data-focused, declarative model will allow us to effectively model the expected outputs from an Agile process, some of which is generated/updated during the simulation, and capture the inherent data-constraints among activities.

In this chapter, we analyzed the types of agent architectures and process modeling notations and selected a combination to support Agile process simulation. Building on these decisions, we constructed an agent-based reference model that describes the constructs and interactions necessary to model Agile processes for simulation. In the next part, we will use this reference model as the basis for our process evaluation environment (simulation); serving as the vocabulary for expressing the cognitive and world models.

Part II

Simulator

With the reference model in-place, we now turn our attention to the simulator which will execute simulation models based on the reference model. Under a decentralized control model, the simulator is composed of the agents that drive the simulation and scaffolding which initializes the simulation from the inputs, starts the simulation, and both collects and emits the results at the end of the simulation.

Agents are the key component in this, yet defining them is non-trivial. As multi-agent systems are composed of many individual agents, we will first define a single-agent simulation (the agent itself in Chapter 5 and the simulation driver in Chapter 6) then scale it up to support multiple, interacting agents (Chapter 7). Throughout this we focus heavily on modeling the cognitive aspects of agents and the sharing of information, including how to construct a domain model that will serve as the basis for composing process executions.

Chapter 5

Defining a Single Agent

Recall that we want to use utility-based, deliberative agents in our simulation because Agile's low process prescription lends itself, broadly, to goal-oriented reasoning while its emphasis on individual behavior (including preference) lends itself, specifically, to utility-based reasoning. All autonomous agents[1] (hereafter, simply agents), regardless of the specific form of reasoning, use a common, high-level architecture that we previously described as the agent eval-loop (Figure 5.1). In this architecture, the world is the shared store of all information or data objects. The agent perceives/observes changes in the world, updating its internal representation of the world; selects an action to perform; then enacts the action, updating the shared world. Thus, actions are effectively transforms from one world to a future world.

By elaborating the phases of the agent eval-loop, this chapter lays out the components of an Agile-capable agent that will serve as the basis of a single-agent simulation.

[1] Herein, autonomous agents are agents with the ability "to act without the intervention of humans or other systems[; wherein the agent has] control both over [its] own internal state and over [its] behavior" [116].

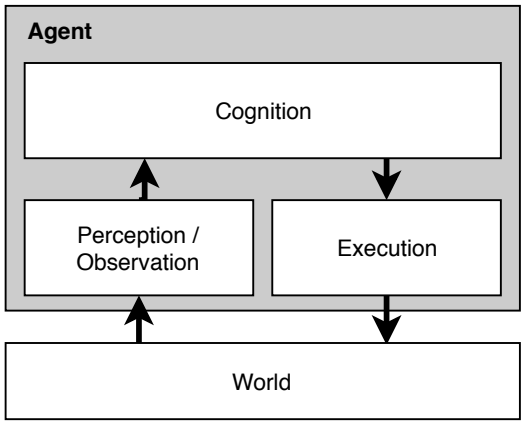

Figure 5.1: The common agent architecture.

5.1 Cognition

During the cognition phase, the agent selects the next action it will perform based on its symbolic representation of the current world (Listing 5.1). The agent composes sequences of process activities (plans) expected to reach a goal state from the current world—a process called planning—and selects the most desirable plan. The agent then adheres to the plan, enacting each action in the plan, until the agent reaches the goal state, the plan is depleted, or there are updates to the world which invalidate the plan. In the latter two cases, the agent forms and enacts a new plan.

Listing 5.1: The agent's action selection procedure.

```
var plan: Queue<Action> // Initially empty

if( isAtGoalState(worldRepresentation) ) {
  // End the simulation
}
else if( isEmpty(plan) or !isValid(plan) ) {
  val candidatePlans = generatePlans(worldRepresentation)
  plan = selectPlan(candidatePlans)
}
return nextAction(plan)
```

5.1.1 Planning

A planning module (planner) within the agent is responsible for constructing processes-compliant plans to be considered for enactment by the agent. Plans, which are a sequence of actions, express a specific flow or execution over the process. If the plan connects the start and goal states, the plan is a complete execution of a process.

AI planners and their use in single- and multi-agent systems have been an area of active research for quite some time [52]. Single-agent planning systems construct plans from the agent's symbolic world representation (a set of data objects, called beliefs), capabilities (the actions it can perform), and goals (future worlds where some desired data properties hold). Scaling single-agent planning to multi-agent systems requires additional considerations for inter-agent constraints and unexpected world changes made by other agents; we will address these in a later chapter.

Terminology and Assumptions A declarative reasoning model, such as a planner, has at least two primitives—actions and goals—both of which are expressed over *data objects*—such as the artifacts and problem reports in our reference model. *Actions*, which are preconditioned on the (non-)existence of desired data objects within the world, transform the current world (set of data objects) into a new world by adding additional data objects to the world. Actions may be independent or belong to multiple processes/fragments. They serve as the basic building blocks for processes. *Goals*, meanwhile, express a set of possible, desirable worlds in which a set of data objects exist.

As we will address later, during planning we assume that each action will bring about its expected outcome. We do this because we cannot reasonably account for every unknown during planning and we can, with relative ease, generate a new plan that accounts for the updated world. Similarly, we assume during planning that enacting an action results in the existence of one or more data objects of each output type. We model this as an action producing exactly one of each type of output (e.g., requirements elicitation produces only one requirement). To do otherwise would require the ability to predict the future with reasonable accuracy and would be unrealistic. Developers/business architects generally cannot predict the exact number of requirements they will generate during elicitation let alone the exact requirements (but, should you encounter someone who can, I'd love to work with him/her).

5.1.1.1 Existing Approaches

Research in other domains strive for outcomes similar to single-agent planning. Situational method engineering (SME) aims to construct situation-specific, fully-specified processes (methods) from process fragments stored in a repository [87]. While there have been attempts to define and classify the different process construction and tailoring approaches (e.g., [86, 45, 19, 73]), all approaches construct a process from the process' goals/requirements (process assembly), from an existing process model/metamodel (transformation), or from some combination of the two.

Assembly Process assembly is a means of composing a process from a repository according to its requirements/goals [45]. Ralyté, Deneckère, and Rolland's assembly approach constructs a process requirement map that is used to retrieve suitable fragments from the repository, according to the fragment interfaces and descriptors, and assess the degree to which they satisfy the requirements (the similarity measure) [86]. Fragments with low similarity may be further refined to improve their similarity [45]. After retrieving all necessary fragments, the fragments are assembled by bridging non-overlapping fragments or by integrating overlapping ones.

Goal-oriented AI planners similarly assemble fragments or individual actions into process executions (sequences of activities called plans) based on goal satisfaction or some heuristic. Planning from existing fragments allows the agent to leverage procedural knowledge in order to reach the goal state [40], while first-principles planning—planning from the set of atomic, enactable actions—provides greater flexibility in that the agent can handle situations that were not originally conceived of by the modeler [11]. That additional flexibility comes at a price, however, as first-principles planning is computationally costly.

Transformation Rather than constructing processes based on their requirements/goals, we can also build processes by transforming—augmenting, reducing, or changing—an existing process model or metamodel [45]. With the intent of generating a complete, enactable process up-front, Ralyté, Deneckère, and Rolland describes several (meta)model transformation strategies—including generalizing, specializing, instantiating, and adapting—that use guides, patterns, and/or fragments in the repository to construct new processes [86]. To support these transforms, Chroust and Lee and Wyner introduce formal semantics for expressing and extending (specializing and refining) existing processes, preserving flow [19, 60].

Similar to the SME approach of constructing a process from a metamodel, the Jason agent interpreter dynamically composes plans from a set of initial goals and plan fragments [11]. Each plan fragment specifies how to achieve a goal through a sequence of actions and goal changes (add/remove). Effectively, fragments are sequences of actions or subgoals that must be instantiated on-the-fly with other fragments (augmentation). This allows agents to leverage both existing procedural knowledge, represented as fragments, and still handle unexpected situations with less computational cost than first-principles planning [41]. In contrast to SME's approaches, which construct complete, enactable processes, Jason and similar frameworks build only what is needed in order for the agent to continue running, allowing these agents to react to changes in the environment.

Plan-construction using existing fragments—as Jason does—limits the agent's behavior to only those action sequences (fragments) defined by the modeler at the outset. With the number of practices (process fragments/procedures) and their variants used in the industry, pre-populating a process fragment repository to account for the potentially-vast practice variance within a team is unreasonable. Applying transformations to these fragments may help in this regard, but not necessarily enough, especially since control-flow must be preserved. Further, composing overlapping fragments is challenging and requires manipulating fragments so the sequences are comparable.

Rather than constructing a plan from fixed fragments, we want to do goal-oriented, first-principles planning—building plans to satisfy one or more goals from individual activities—and seek optimizations from there. Together with a declarative (data-flow-based) view of processes, first-principles planning allows us to model practices that were not preconceived by the modeler but still enactable by the agent. As stated, this can be computationally expensive. To mitigate this, we plan to divide larger goals into sets of subgoals that the agent can complete.

5.1.1.2 Planning Domain[2]

In order to construct plans and satisfy goals, agents require a planning domain model—the set of actions an agent may perform and the sequencing constraints over those actions. Unfortunately, there are no existing, declarative planning domain models nor guidance for creating them in the literature. The objective of this section is to provide guidance for creating a planning domain

[2]Much of the content of this section appears in an earlier published work by the author ([30]). It is reproduced here in compliance with any existing agreements and applicable law.

model for agent-based simulation.

What are the properties of a suitable planning domain model? For agents, planning is performed in situ, and planning failure results in simulation failure, thereby wasting valuable analysis time. These failures occur if no valid plan can be constructed in the planning domain. To prevent this, we desire a planning domain model where we cannot construct an invalid plan if the planner cooperates in proactively ensuring plan validity. A plan, which is essentially a narrowly-defined process, is valid if it is both internally-complete (all activity dependencies are present in the plan) and internally-consistent (for each plan activity, all activities it depends on precede it in the plan) [44].[3] Thus, with a cooperative planner, invalid plans are only generated when the domain model contains internal incompleteness or inconsistency. We, therefore, desire a declarative planning domain model that is both internally-complete and -consistent.

As a step towards multi-agent, agile-process simulation, we prescribe a method for constructing an internally-complete and -consistent (ICC), single-agent[4] planning domain model using iterative elaboration. We show that, given an initial ICC domain model and ICC process fragments, each elaboration produces an ICC domain model. Further, we provide guidance for specifying achievable intermediate goals, despite possible activity output non-determinism, within the model.

Overview A process (or fragment) is a sequence of activities (data transforms) executed by an actor to some end. These activities consume and produce data objects (artifacts and resources), which make up the agent's world model at a given point. Data objects may be related to each other (e.g., by composition). Further, processes contain initial and goal states that are satisfied by one or more worlds.

Software development can be expressed as a process with a single activity: transforming a problem statement into software that addresses the problem. However, this lacks an operational description. We wish to iteratively elaborate this, our initial domain model, until we have enough detail to express all processes under consideration and provide alternative actions to the agent. Existing process repositories—literature and fragment repositories (e.g., [47, 100])—contain a wealth of method information that can aid our elaboration. Using these as fragment sources, we have summarized our method in Listing 5.2.

[3]There are other situation-independent process quality criteria [44, 45]. However, they address process desirability rather than validity, and are unrelated to plan formation.

[4]We assume, for now, the agent has perfect knowledge of the world.

Listing 5.2: Our approach expressed algorithmically.

```
var domainModel: DomainModel // Initialized with an ICC set of activities and constraints
while( !shouldStop(domainModel) ) {
  val (activity, fragment) = locateReplacement(domainModel)
  val generalizedFragment = generalize(fragment)
  domainModel = replace(domainModel, activity, generalizedFragment)
}
```

We begin by selecting an activity to replace within the domain model and identifying a fragment from the repository that preserves the dependencies in the domain model; specifically, a fragment whose initial state is a subset of the activity's inputs and whose goal state is a superset of the activity's outputs. Further, the fragment must not contain an activity that removes data as part of its transformation.[5]

Next, we transform the fragment; generalizing it by removing all constraints (e.g., control flow constraints) except those inherent to its constituent activities (their data dependencies). As we will show, this allows agents to combine activities in ways that are valid, but may not have been captured in the process repository. We then replace the previously selected activity with the generalized fragment; resulting in a new ICC domain model.

We repeat this process until we have a model that can express all of the processes that we wish to evaluate using simulation and we have enough detail to provide alternative actions to the agent that are not specified in the process.

Ensuring Domain Model Suitability We wish to construct a domain model such that, given an initial ICC domain model and a set of ICC process specifications, the iterative elaboration of the domain model will also be ICC. We will show that the generalize and replace transforms preserve the process/process fragment specifications' and domain model's ICC properties.

Definitions A *process fragment specification* is a set of activities and sequencing constraints. Similarly, a *domain model* is a collection of activities, sequenced according to their dependencies, with initial and goal states to define/limit its scope. For model simplicity, we assume activities non-destructively consume data objects. Further, we omit input/output object cardinality from the planning domain model and leave it to activity execution.

[5]Rather than removing data objects, we model removal of artifacts as changes to scope information kept by the agent.

In the planning domain, an *execution* is a non-empty sequence of activities. An execution is *valid* if it is ICC. A valid execution is *goal-reaching* (or *domain-model-goal-reaching*) if it connects the initial and goal states.

A domain model is *internally complete* if, for each activity in the domain model, its inputs are generated by another activity within the model. A domain model is *internally consistent* if, for a given activity, there is a sequence of activities from the initial state that provide the inputs required by the activity. Thus, a domain model (or, correspondingly, a process specification) is ICC if it is composed of a set of activities and sequencing rules such that the sequencing rules together with the activities make up a non-empty set of goal-reaching executions and every activity within the model lies on at least one execution.

To simplify our discussion, assume the domain model includes an activity with no inputs to produce the initial state (v_{start}) and another that consumes the goal state with no outputs (v_{end}). If more than one goal state exists, define $v_{(end,1)}, ..., v_{(end,m)}$ such that these nodes generate a token artifact (indicating the goal state has been reached) that is consumed by v_{end}. Do a similar thing if there are multiple initial states. Thus, without loss of generality, assume there is one v_{start} and one v_{end}.

Generalization For a given ICC process fragment specification, P, with a set of activities V_P, we generalize P (call it $G(P)$) by removing all constraints except the innate dependencies of its constituent activities (the data dependencies). Here, we show that the result of this transform is an ICC generalized fragment.

Let $\mathcal{P}(V_P)$ be the set of all executions (valid or not) over the activities in P and including v_{start} and v_{end} where each of the sequences begins at v_{start} and terminates at v_{end}. We produce $K(V_P)$ by removing all executions from $\mathcal{P}(V_P)$ where the dependencies of an activity do not precede it. Thus $K(V_P) \subseteq \mathcal{P}(V_P)$ is the set of all executions within $G(P)$.

To show that $G(P)$ is ICC, we must show that $K(V_P)$ is non-empty; $K(V_P)$ contains only valid, goal-reaching executions; and every activity in V_P is in some execution in $K(V_P)$.

Let $e(P)$ be the set of all goal-reaching executions in P. Because P is ICC and by the construction of $K(P)$, we know that $e(P) \subseteq K(V_P)$, $e(P)$ is non-empty (it contains P), and $e(P)$ contains all activities in V_P. Further, because $K(V_P) \subseteq \mathcal{P}(V_P)$, $K(V_P)$ contains no activities besides those in V_P. Thus, $K(V_P)$ is non-empty and every activity in V_P is in some execution in $K(V_P)$.

In the construction of $K(V_P)$, we removed all executions from $\mathcal{P}(V_P)$ containing an activity whose dependencies do not precede it in the execution. Thus, each execution in $K(V_P)$ is ICC. Further, since each execution in $\mathcal{P}(V_P)$ begins and terminates at v_{start} and v_{end} respectively, each execution is a valid, goal-reaching execution.

Since $K(V_P)$ is a non-empty set containing all valid, goal-reaching executions within the generalized fragment, $G(P)$, and every activity is contained in at least one execution, we know $G(P)$ is ICC.

Replacement Generalization leaves us with both an ICC domain model, D, and a generalized, ICC process fragment, $G(P)$. We wish to compose them into a new domain model.

Previously, we selected a process fragment, P, such that, for some activity $a \in D$, the fragment's initial state is a subset of the inputs of a and the fragment's goal state is a superset of the outputs of a. By the construction of $G(P)$, P and $G(P)$ have the same initial and goal states. As $G(P)$ is ICC, its inputs are provided by a's dependencies, and its outputs satisfy a's dependents, $G(P)$ can replace $a \in D$ and the result is a new ICC domain model, D'.

Additional Properties for Simulation Simply being able to generate plans from an ICC domain model is not enough for simulation. In this section, we show both that agents can replan on-the-fly using this model, and that, with guidance, modelers can specify other achievable goals.

Replannability Thus far, we have ignored output non-determinism: an activity may produce one of multiple output sets upon execution (e.g., a test-run may emit a success message or fail, providing an error report). This has no bearing on the ICC of the model; however, it does trigger replanning. We want to ensure that the agent can still generate a valid plan when starting from the current world (*replannability*).

Assume that we have an execution, e, beginning at v_{start}, that led us to the current, non-goal world. Because artifacts are never removed from the world, once we execute an activity, all of its outputs are available from that point on. Thus, we must supplement e so that it reaches v_{end}. Because the domain model is ICC, there is at least one goal-reaching execution, ϵ, in the model. By removing all completed activities in e from ϵ (call it f) and appending f to e, we know the result will be a goal-reaching execution as it complies with the dependency constraints and connects v_{start} and v_{end}. The sequence, f, is the agent's new plan to reach the goal.

Thus, from any world reached by performing a valid sequence of activities, we can create a plan from that world to the goal state.

Intermediate Goal Planning By replannability, we can reach the domain model goal state from any reachable world; however, some teams may only be focused on achieving some intermediate goal state. We'd like to define other, intermediate goals and plan to them. This would allow us to both support both simulations with different end goals and subgoal-based planning.

Intermediate goals (IGs) are worlds in which desired data objects exist. When all activities are deterministic, an IG is expressible as a non-unique set of activities that produce the IG state. Using the same technique used to show replannability, we can generate a plan to reach each activity comprising the IG. Since activity output may be non-deterministic during plan enactment (e.g., the tests fail), we must express IGs only over those data objects that we could arrive at deterministically (without relying on exceptional activity output). Is there a way for us to treat non-deterministic activities as deterministic for planning purposes?

Activities that produce both expected and exceptional output sets result in output non-determinism. Exceptional activity output in the planning model stem from expected exceptions during activity enactment (e.g., a defect in compiled code) or imperfect world knowledge (e.g., an artifact that is unexpectedly missing).[6] The latter case is not possible for a single agent with perfect knowledge; however, even with additional agents, missing knowledge can be gathered by communicating with others or exploring reality. Because the domain model is shared among the agents, the resultant, updated world is also the product of some sequence of activities within the domain model and can be planned from. Thus, we must only consider exceptional output stemming from errors.

To ensure IG reachability, we want to prevent our IG state from including any artifact that can only be reached through an unexpected output of a non-deterministic activity. As illustrated by the `run tests` activities of test-driven development (TDD; Figure 5.2), the expected output depends on the world's state (e.g., the tests can only pass if the increment's code is present). To deal with this sort of non-determinism, some planners specify policies (over a control-flow-based planning domain) directing the planner to select a specific action when in a given state [42]. We could similarly guide the planner by specifying expected output based on the current state. In the TDD example, such a policy would expect test failure if the implementation is not present

[6] Several forms of workflow faults exist [95]; however, in terms of expected activity output, only these apply.

and test success if it is. This makes planning deterministic, and allows us to include additional data objects in our definition of an IG state.

5.1.2 Selectors

After generating plans, the agent must select one to enact. As we discussed earlier, there are a number of ways to do this. Because we want to capture individual preferences, we use utility-based reasoning to select the most desirable plan from the set of candidate actions (Listing 5.3). To do this, we must quantify the desirability (utility) of each plan.

Listing 5.3: The agent's plan selection procedure.

```
fun selectPlan( candidatePlans: Collection<Plan> ): Plan {
    val plansWithMaxUtility = candidatePlans
        .groupBy { plan -> computeUtility(plan) } // Computes and groups the plans by utility
        .maxBy { entry -> entry.key }
        .value

    return selectRandomly( plansWithMaxUtility )
}
```

Utility functions provide a means for quantifying the desirability of plans. This desirability must capture both *what* the agent wants to achieve and *how* it will achieve it. For example, developers may choose a project because it provides a means for them to learn and develop new, marketable skills. At the same time, they may also have habits or preferred ways of achieving those goals such as pair programming. These desires may be personal or they may be part of the process.

To facilitate and simplify utility computations we use goals and sub-goals to define units of work and demarcate desirable states—sets of worlds that the agent may want to bring about. As such, goals effectively subdivide the planning space and allow the agent to reason at a higher level. However, simply including goals is insufficient for our needs. We need to know both when a unit of work begins and ends in order to do some forms of reasoning, such as minimizing work-in-progress. To achieve this, we can mark the goal (really, the scope in which the goal is achieved) as started when the world is in a certain state, immediately before an action(s) is enacted, or during an action's enactment. In the latter two cases either the goal-start is marked immediately prior to downstream actions or during the first action that contributes to goal satisfaction. Rather than tie the process-agnostic domain model to a specific process, we instead

introduce a means for applying these limited control-flow constraints on the process at plan time and use them as markers for the utility function.

Once we have computed the utility of each plan, we want to select the most desirable plan. In the event of a tie for the most desirable plan, we randomly select from among them because, from the agent's perspective, they are all equally good.

5.1.3 Theory In Practice: Simulating TDD

Having presented and supported an approach for constructing a domain model, we use our proposed approach to generate a domain model and show, in a simple simulation, that

1. the domain model can be used to generate valid plans and
2. agents can reach the goal even with a lightweight process specification.

This simulation does not attempt to model a complete team, and, therefore, does not satisfy all of the aims of this work. Rather, this simulation is a proof of concept illustrating how we can use a domain model to dynamically compose process executions. We will build off of this in later chapters.

5.1.3.1 Constructing a Planning Domain Model

Having defined an approach for generating a domain model, we want to put it into practice, applying it to create a domain model suitable for expressing test-driven development (TDD; Figure 5.2).

Initial Domain Model Begin with an initial domain model: `software development`. Of the numerous possible elaborations, we select the V-model [104] since TDD is focused on testing. We expand `implementation` using IEEE Standard 1074 [47] and introduce activities for declaring, implementing, and integrating interfaces as prescribed by TDD for testing [100]. Spillner, Linz, and Schaefer provides a general testing process that elaborates each of the testing activities [104]. To support TDD's test automation, we replace `test coding` with the implementation process from earlier and `test execution` with TDD's definition [100]. Finally, we introduce refactoring according to Fields, Harvie, Fowler, and Beck's description [37]. Table 5.1 shows a subset of our domain model.

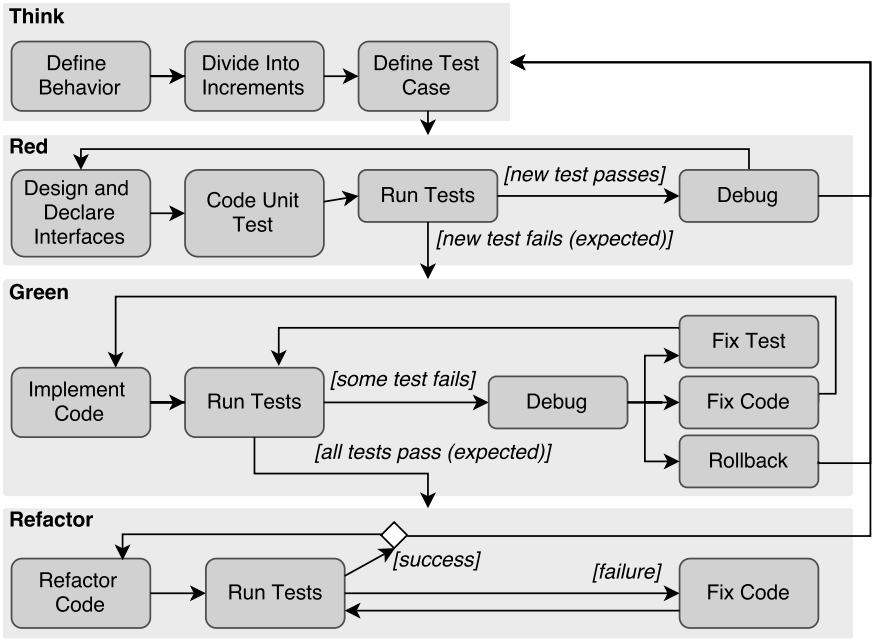

Figure 5.2: TDD – Detailed Control Flow [100]

Table 5.1: Domain Model Subset – Inputs and Outputs

Activity	Input Artifacts	Output Artifacts
Define Requirements	Concept	Requirement
Technical System Design	Requirement	System Architecture
Transform Architecture to Components	System Architecture	Component Definition, Component Integration Model
Test Planning	Requirement	Test Design
Specify Component	Component Definition	Component Design, Increment Requirement
Declare Interface	Increment Requirement, Component Design	Increment Interface
Implement Increment	Increment Requirement, Increment Interface	Increment Implementation
Integrate Increments	Increment Implementation, Component Design	Component Implementation

Table 5.1 Continued: Domain Model Subset – Inputs and Output

Activity	Input Artifacts	Output Artifacts
Integrate Components	Component Implementation, Component Design	System Implementation
Compile System	System Implementation, Compilation System	Compiled System
Compile Interface	Increment Interface, Compilation System	Compiled System
Specify Unit Tests	Increment Requirement, Increment Interface, Test Design	Logical Unit Test
Generate Concrete Unit Tests	Logical Unit Test	Concrete Unit Test
Implement Unit Test	Increment Interface, Concrete Unit Test, Testing Tool	Automated Unit Test Script
Integrate Test	Automated Unit Test Script, Test Design	Test Suite Implementation
Compile Tests	Test Suite Implementation, Test Compilation System	Compiled Test Suite
Execute Tests	Compiled Test Suite, Compiled System	Test Execution Result

Model Validation Before simulating, we want to verify the generated model can express TDD. To check this, we mapped the different portions of TDD to our generated model (Table 5.2).

Table 5.2: Mapping from TDD to the Domain Model Subset

TDD Activity	**Domain Model Activity**
Define Behavior, Divide into Increments	`Specify Component`
Define Test Case	`Specify Unit Tests`, `Generate Concrete Unit Tests`
Design & Declare Interfaces	`Declare Interface`
Code Unit Test	`Implement Unit Test`
Run Tests	`Execute Tests`
Implement Code	`Implement Increment`

One benefit of our approach is that we can generate activity sequences that were not considered during the construction of the domain model. For example, the activities in the generated domain model

can be easily used to specify test last development (TLD).

5.1.3.2 Simulating

Having constructed an ICC domain model, we demonstrate its use with a simple simulation.

Simulator Set-up To model an actor working to develop and unit-test a feature, we wrote a single-agent simulator. In it the agent deliberates—forming a plan—and executes the planned activities, simply producing the plan-predicted activity output.

Deliberation Our planner generates all possible plans from the data model by forward-chaining activities to a fixed depth (here, three). Plans are rank-sorted according to a utility function and the plan in the first position (the highest utility) is selected, even if there is a tie. On plan completion, the agent replans. This repeats until the agent reaches the goal state.

Further simplifying the planner, we omitted rework and non-deterministic activity outputs from the domain model. We will address them in a later section.

Utility Utility functions help the agent determine plan desirability. To capture process adherence, we modeled the process as a set of partial orderings—pairs capturing a succession relationship: one activity must come after the other—and biased behavior towards quickly completing orderings with the following utility function:

$$U_i = \frac{2}{3}W_i + \frac{1}{3}U_{i+1}$$

where i is the position of an activity in the plan (where position 0 is the first activity in the plan), U_i is the utility of the activity at position i in the plan, and W_i is the weight of the activity at position i based on initiating or completing an ordering (arriving at a node on the left or right side of a pair, respectively); defined as:

$$W_i = \begin{cases} 1 & \text{if the activity at } i \text{ completes an ordering} \\ 0.5 & \text{if the activity at } i \text{ initiates an ordering but does not complete one} \\ 0 & \text{if the activity at } i \text{ does not initiate or complete an ordering} \end{cases}$$

This utility function computes backwards, from the last element in the plan to the first. Thus, for a plan of length l, we begin computation at U_{l-1} (with $U_l = 0$). The utility of the entire plan is U_0.

To contrast process-compliant behavior, we also model random behavior (or behavior with no process constraints). To do this, we defined a constant-value utility function. Under these conditions, the agent

should choose plans at random until it reaches the goal.

Within the simulation, we also added in two system functions: IDLE and EXIT. The former represents a no-op for the agent while the latter causes the simulation to terminate. We biased both utility functions to favor exiting as soon as possible and to avoid idling using the following function:

$$F_i = \frac{1}{2}(U_i + B_i)$$

where

$$B_i = \begin{cases} 1 & \text{if the activity at } i \text{ is EXIT} \\ 0 & \text{if the activity at } i \text{ is IDLE} \\ 0.5 & \text{otherwise} \end{cases}$$

Thus, for the process-adherent utility function, we have the following biased utility function:

$$F_i = \frac{1}{3}W_i + \frac{1}{6}F_{i+1} + \frac{1}{2}B_i$$

Results We ran the simulation three times with the same goal: once each for no process (constant utility), fully-specified TDD, and fully-specified TLD. The resultant executions (Listing 5.4) indeed show that the agent was able to use the domain model to plan and execute TDD and TLD. Further, it generated a valid plan to reach the goal even without a specified process.

Even though we were able to demonstrate our method's use in planning and simulation, this controlled experiment was done with a simple example, leaving out much of the complexity discussed earlier (specifically the non-determinism and rework). In the upcoming sections and chapters we will extend this work to both support multi-agent simulation and activity enactment.

Listing 5.4: Simulations by Utility Function

(a) Constant-value

defineRequirements
technicalSysDesign
testPlanning
trans_ArchToComp
specifyComponent
declareInterface
compileSys
implIncrement
specifyUnitTests
integrateIncrements
integrateComponents
genConcreteUnitTests
implUnitTest
integrateTest
compileTests
compileSys
executeTests

(b) Weighted: TDD

defineRequirements
testPlanning
technicalSysDesign
trans_ArchToComp
specifyComponent
declareInterface
compileSys
specifyUnitTests
genConcreteUnitTests
implUnitTest
integrateTest
compileTests
executeTests
implIncrement
integrateIncrements
integrateComponents
compileSys
executeTests

(c) Weighted: TLD

defineRequirements
technicalSysDesign
trans_ArchToComp
specifyComponent
declareInterface
implIncrement
integrateIncrements
integrateComponents
compileSys
testPlanning
specifyUnitTests
genConcreteUnitTests
implUnitTest
integrateTest
compileTests
executeTests

5.2 Enactment

Once the agent has selected an action, the agent must enact (execute) it to bring about changes in the world. These are changes to reality rather than the symbolic representation of the world. However, things do not always go as expected. Beyond addressing quantities of expected outputs, things may go wrong during enactment. For example, in real life, developers find (and inject) bugs while coding. We want to model this mismatch between expected outcomes and actual results within our simulation.

Outputs By default, each activity produces only its expected output. A modeler may override this behavior and specify a specific enactment model per activity within the domain model. These enactment models can be deterministic (like the default output), stochastic (e.g., injecting latent faults or other problems according to a random distribution), or some mix of the two. However, we constrain the enactment model's outputs to only those artifacts that are specified in the domain model as the activity's outputs, problem reports (that will reside in the agent's belief base), and faults/problems (that the agent is unaware of and reside only in reality). Further, if the enactment model does not produce at least one artifact of each type specified in the expected output, the enactment model must produce a problem report. This limits the ways in which an action can deviate from the domain-model-specified

outputs in order to ensure replannability.

How does it ensure replannability? As rework is detected, the agent produces problem reports describing detected faults/problems (we will simply call these faults) that trigger the re-execution of the activity that produced it. This re-execution produces a new version of the faulty artifact(s). This, in turn, will trigger the recreation of downstream artifacts generated based on the faulty artifact (we track artifact provenance by mapping the specific inputs that led to the generated outputs). This will continue until either the activity that produced the rework-triggering problem report is rerun with new versions of its inputs or a different sequence of activities is chosen that skips the problem-detecting activity.

Time In addition to modeling the data objects generated by the action, we must also model the time required to generate them (the enactment time). In a typical agent-based system, agents operate in the physical work. Because it can take some time for the agent to complete the action, the agent initiates the action as soon as it can. When the action is complete, the agent perceives the world, replans if needed, and enacts another action. In a simulation, however, we cannot do this. There is no natural time to complete an action. We do not need to wait for network controllers or actuators before observing the results of our actions. We just generate new data objects. This leads to unrealistic time metrics from the simulation. Instead, we need to determine, upfront, the amount of time it will take to complete an action—the enactment time—and wait to continue the eval-loop until either the action is fully enacted or it is interrupted.

Since we are generating new data objects during enactment, when are those data objects added to the world? We do not want to do this at the beginning of the enactment period because, when we increase the number of agents, others might be able to respond to these changes before the agent is "done" with them. If they are emitted throughout the enactment period, we need to model both the order and time required to produce each output, a task that seems both infeasible and unnecessary. Emitting the new data objects at the end of the enactment period side-steps both issues, but, if the agent is interrupted (and the work is abandoned), the agent will not emit anything. However, since Agile projects follow the 0/100 rule of earned value management (no partial credit is given for work; credit is only awarded when work is complete) [83, 22], we do not plan to model abandoned work.

It is worth noting that determining the enactment time is entirely distinct from how long an agent might *think* an action would take. The former addresses the actual time used in the simulation to complete the task (the actual duration required) while the latter is an estimate that the agent (or other agents) strives for and, as such, is part of the agent's cognitive reasoning.

5.3 Perception

After updating the world, the agent perceives changes to the world and updates its beliefs—its symbolic representation of the world. The ability of agents to perceive vary from complete or perfect perception (its symbolic representation perfectly captures the state of the world) to little or no perception/observability. Since we want to approximate developers, we want partial world observability. As projects scale, developers cannot keep track of what has been done in other parts of the project. There is just too much there. Instead they ignore irrelevant parts or rely on abstractions of the parts that matter. But, even in small, single-developer projects, developers still only have an imperfect perception of reality. For example, while writing code, a developer may inadvertently inject a fault into the logic of the system. If the developer had full observability, he would be able to immediately identify the fault and fix it, rendering verification useless. Instead, developers use various verification techniques to make latent faults visible so action can be taken. Partial perception will become more important later when we scale up our simulation to include multiple agents.

Within our simulation, we effectively combine the perception and enactment steps when there is a single agent. The agent both creates and perceives its own creations, updating its beliefs as soon as it generates artifacts. There are, however, some action outputs that are not known by the agent at creation time. In particular, faults remain unknown to the agent until the fault triggers the creation of a problem report.

Chapter 6

Simulation Driver

A single-agent is, by itself, insufficient for simulation. The agent still requires the world in which it is situated and some (shared) notion of the progression of time. In this chapter, we define the simulation driver—the component that progresses the simulation and enables agent interaction—that allows us to simulate Agile development (depicted in Figure 6.1). Because we want to ultimately scale this beyond a single-agent, this section will describe a simulation driver suitable for both single- and multi-agent simulation.

Progression Earlier, we discussed the different representations of time within simulations and selected a discrete representation in order to model distributed control and individual behaviors. Yet, without some shared resource, agents may advance time at different rates. In the real-world, time is shared, and, in order to ensure meaningful results, agents within our simulation must also have a shared timer. The simulation driver provides this.

Because time progresses in discrete steps and only when events occur, the driver should not only maintain the timer, but advance it. The driver can only do this with the support of the agents within the simulation. In our simulation, the driver asks the agents how much time they will need for the next work session. Agents plan out their next actions and submit a bid to the driver equivalent to the amount of time they will need to complete their next planned action. The driver selects the lowest bid as the amount it will use to advance the timer. Recall that the agent bids are the actual time it will take the agent to complete its selected action, not an estimate of how long the agent thinks it might take. This is an important distinction as the bid controls simulation progression by serving as a proxy for executing the action in the physical world, while effort and duration estimates are a product of the agent's cognitive processes and may factor into its decision-making.

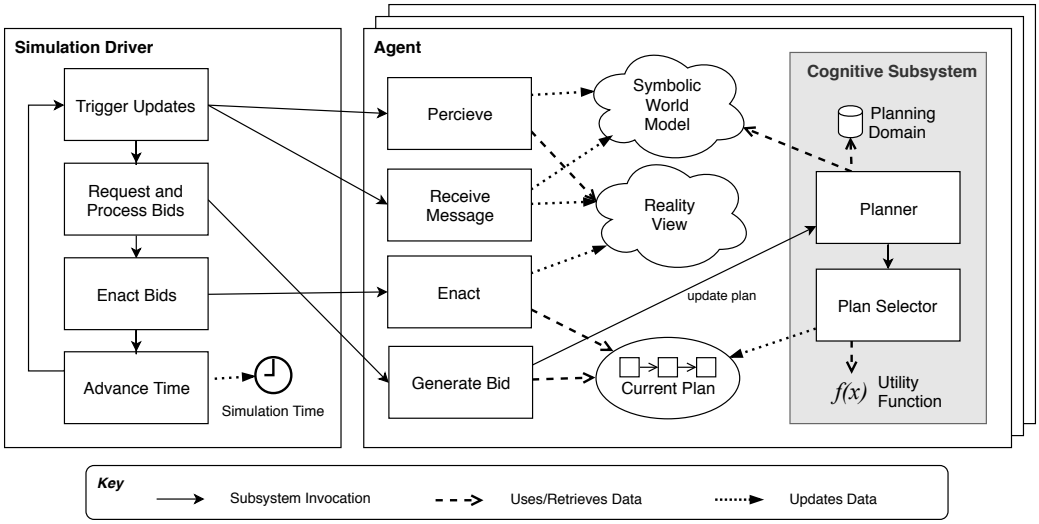

Figure 6.1: Interactions between the simulation driver and agents.

After the bidding process is complete, the driver allows each agent to enact their selected actions for the given amount of time, updating the world on completion. If an agent cannot complete an action at the end of the allotted time, its progress is saved as a belief in the agent's belief base to be completed during a future time allotment. This gives us the opportunity to model interruptions, where an agent's attention is pulled to other things, and the resultant productivity losses [64, 31].

After each agent has completed their work for the allotted time period, the timer is advanced and the process repeats until the simulation reaches a goal state or triggers another terminating condition (e.g., project abandonment due to cost overruns). Because this requires goal/condition processes, something that agents do as part of the eval-loop, it does not make sense to duplicate the functionality in the simulation driver. Instead, we will use some agent—not necessarily a developer proxy, but rather a control agent—to perform the goal/condition processing.

Enabling Agent Interactions The simulation driver is also responsible for enabling and managing the interactions among agents, both indirect and direct.

Agents interact with each other indirectly by updating the world in which they are all situated. When one agent produces an artifact, other agents can observe and respond to it (e.g., change their plans). The shared world (the source of truth) can be located either within the driver or within some control agent. If the driver owns the shared world, the driver must address resource contention issues as they arise. Instead, if we treat the shared world as simply a set of facts (true beliefs), we can use our earlier control agent, the one that assesses termination conditions, to maintain the shared world and

serve as the source of truth. This allows us to reuse existing structures (e.g., the belief base) to maintain the state of the world and normalize all interactions among agents. The driver would then maintain connections between the control agent—call it the source-of-truth (SoT) agent—and every other agent in the simulation. When an agent is ready to update the world, it transmits its updates as messages to the SoT agent over this connection. An agent "observes" the world by either requesting updates from the SoT agent ("pull" updates) or by receiving updates from the SoT automatically ("push" updates).

In this discussion, we assume there is a means for direct interaction (communication) among agents. As this is external to the agent, the driver is responsible for providing communication channels for agents. In order to perform updates with the SoT, we require at least a low-latency channel. However, there are numerous methods for communicating with others, ranging from instantaneous to delayed delivery and sporting features like push or pull notifications, each with varying effectiveness and disruptiveness. As low-latency collaboration is a key part of Agile processes, we want the ability to model both the impacts of the different communication channels and the agent's reasoning over which channel to use. In terms of simulation progression, delayed communication channels can generate events that necessitate a shorter simulation-time advancement than the bids submitted by the agents, thus the driver must also consider delayed communication when considering bids to advance the simulation clock.

Chapter 7

Scaling-up: Multi-Agent Simulation

Individuals rarely develop software alone. Most projects require multiple stakeholders to achieve the goal, using processes to manage interactions and simplify/reduce non-value-adding work. As we want to evaluate processes prior to adoption, we want to extend our single agent simulation environment to support workload distribution over multiple agents.

Collaborative mutli-agent systems, such as the one we are describing here, are indeed similar to distributed systems in that each agent is an autonomous unit of computation connected to other agents over some medium or network—in our case, the communication channels managed by the simulation driver—that appears to others as a single coherent system [106]. As such, multi-agent systems have similar challenges including data distribution and consistency as well as potential resource contention.

Data Distribution Setting aside Beck's Pair Programming and its derivatives [9], developers do not share their work constantly. The effort required to do this at scale would stymie an organization as it would create profuse interruptions that would hinder productivity. In an attempt to minimize interruptions, development teams structure interactions using both processes and tools; processes to prescribe both the means of interaction and its timing and tools to facilitate these interactions. Developers hold on to information, especially partially completed work, and only share it at prescribed points. To mimic this within our simulation, agents maintain their own "view" of the world—a local copy of the world that the agent can operate on—until they reach a synchronization point—a point where they update the shared world. From a data perspective, the agent views are *eventually consistent* with the shared world.

As we discussed earlier, agents also maintain a belief base—the set of beliefs about the state of the world—updating it based on observations of the world and communication. Thus, an agent

may have beliefs about the world which are not reflected in either its view or the shared world. For example, during a status meeting or a daily stand-up (Scrum meeting), agents share information about their current/completed work. This work may be local to the reporting agent's view, but, because it communicates about the work, the other agents can generate beliefs about that work.

At pre-defined points in the process, the relevant parts of the individual view must be shared with other agents by updating the shared world. However, this can lead to conflicting updates from stale views or resource contention (for example, multiple claims on the same locks). Within our simulation framework, the SoT agent is responsible for resolving these issues. This allows the resolution strategy to be specified by the modeler (user).

When should view synchronization occur and what should we share? Notionally, we want to share completed work, but this can vary widely by team. Not only do Agile teams define the completion criteria (definition of done) for themselves, some teams even have multi-phase completion criteria where they, for example, share development-complete, untested work at the end of the first phase then do system-level testing in a subsequent phase [25]. As the completeness criteria and other synchronization triggers are defined as part of the process, the modeler must determine when synchronization occurs as part of the process definition input to the simulation.

Still, what we share is not trivial. It does not make sense for us to share everything as this allows us to share incomplete work (which can adversely affect other agents). Instead, we want to share only a completed unit of work and those artifacts generated in its creation. To do this, we need the start and end points of the activity sequence where the work is performed and, because of potentially unrelated activities woven into the fragment, a means for tracking which artifacts relate to the work we will share. Goals provide a natural place to indicate when work is complete. To complement goals and establish a scope of work for them, we track when the scope is entered according to criteria established by the modeler. A scope could be entered by executing one or more actions or reaching a specific world state. By also tracking the provenance of each artifact (the artifacts used to create it as well as scope start markers), we can start to determine which artifacts belong to a scope. However, simply tracking provenance is not enough information to determine inclusion within a scope as input artifacts can originate from different scopes. To address this, each activity marks its output artifacts with a scope based on the scopes of the input artifacts.

We observe that there are three types of actions: splits, merges, and direct transforms. Each of these types take in artifacts and produce new ones; however, splits represent taking an artifact that belongs to a single scope and splitting it into several, potentially parallel, processing units (scopes)—like dividing a story into several increments that can each be developed independently—while merges join these independent pieces into a single new artifact (returning to the "parent" scope)—like integrating

those increments and their tests back into the product. Direct transforms are neither splits nor merges and do not generally trigger scope changes. By tracking individual artifact scope, we can track which artifacts should be shared when synchronization occurs.

Contention With synchronization, agents can share the result of their work. However, this prevents multiple agents from working on the same work product, or, more generally, acquire exclusive use of finite resources such as test environments. To address this, agents acquire locks from the SoT agent prior to executing an action using lockable resources/artifacts. These artifact locks are only advisory, rather than mandatory, as agents may choose to ignore the claims of others and duplicate work. This allows us to model both compliant personalities that respect work claims and prima donna personalities that run roughshod over others on the team because they believe they can do it better.

Even with locks, it is possible to have merge conflicts. Say we have an artifact, such as the product under development, that is composed of other, smaller artifacts. If two agents both update their local copies of the artifact, each adding a new and different artifact to the set, then, when they try to synchronize with the SoT, they will need to merge their respective changes. Here, the SoT is responsible for orchestrating the resolution of the merge conflict by either resolving it itself or by directing the agents to resolve it.

In our implementation, the first conflict-free update the SoT will be accepted. Conflicts will need to be resolved by the agent attempting the update. Similarly, locks are awarded on a first-come, first-served basis. Even though this could lead to agent starvation (an agent with nothing it can do because all available locks are held), we want to allow this so we could model the both the effects of "greedy" agents and agent negotiation on the process.

Chapter 8

Simulation Inputs: Revisited

Throughout the last few chapters, we have striven to define an agent-based simulation suitable for capturing Agile processes and, in the process, we defined the specific simulation input parameters. Agents require knowledge of the set of possible actions they can perform to support planning. More specifically, they require both a domain model that describes the possible actions within their environment as well as actions to aid in coordination, such as acquiring/requesting locks.

Of course, in order to evaluate agile processes, the process must be provided to the agent as an input to the simulation. Earlier, we discussed processes as a set of activities and sequencing constraints on the otherwise unbounded set of expressible activity sequences. These activities are directed towards some goal. Both the constraints and goal-orientation manifest as activity sequence preferences. As such, we express these preferences as utility that direct agents towards process-compliant behavior, such as completing work (satisfying a goal) as fast as possible, maximizing the number of running-tested features [50], or limiting batch sizes or work-in-progress [46], rather than non-compliant behavior. Indeed, individuals drive project success [21] and their preferences should be captured within utility functions. However, process compliance is only one of many preferences held by developers. Other factors, such as personality, comfort, familiarity, or even contrarian views could impact a developer's inherent utility assessment and result in non-compliant behavior [21, 18]. While we proposed a generic solution for using utility, developing a utility function to capture all of the different aspects of preference is a large task that is best addressed in later work. In this work, we limit ourselves to utility functions that drive process compliance.

The impact of the individual manifests itself, not only in the preferences as captured in the utility function, but also in determining action outcomes. This includes traits of the individual such as skills/aptitude. Given that the planning domain model is, by design, too idealized to model action

outcomes stemming from a particular agent, enactment models—models describing the results of action execution in terms of time and world changes—are necessary to translate selected actions into effects that can be perceived. The resultant generated artifacts are shared at process-defined points. Further, the process defines the coordination mechanism to diminish resource contention and ensure effort towards goal-satisfaction is not duplicated.

Thus, in order to simulate Agile processes within our simulator, the modeler must provide the domain model, actions to handle coordination/contention, process scopes defined within the domain, one or more utility functions, activity enactment models for the domain, and activity time-models (expected simulation time) for the activities in the domain. In the next part, we will construct the specific inputs required for experimenting with our simulator and show that the simulator satisfies our requirements for simulating Agile processes.

Part III

Experiment

Chapter 9

Experiment Set-up

The objective of this work is to show a means for evaluating candidate Agile processes prior to adoption. In this section, we will define a set of experiments that show how our simulation framework achieves this.

Expanding on the Agile values and principles defined in the Agile Manifesto [66], we defined a number of requirements for simulating Agile processes that we wish to embody in our simulation framework (Section 2.4). When describing the construction of our framework—the Lean/Agile Process Simulation Environment (LAPSE)—we illustrated how we satisfy these requirements. To further highlight that we satisfy these requirements we wish to show that agents:

1. exhibit process-compliant behavior without a complete process prescription
2. can select different actions based on preference (as we model this through utility, we need to show the impact of utility on the agent's decisions)
3. can communicate and interact with each other to share knowledge
4. can respond to change/rework as well as unexpected action outcomes.

The first two show that LAPSE can model individual self-determination and can handle Agile's lightweight nature; the third shows it can capture communication and interactions among actors; and the fourth shows responsiveness to change or the unexpected. Because this work focuses on the cognitive and perceptive aspects of the simulation, we do not demonstrate that we can model sustainability. However, we believe that it can be modeled within LAPSE with appropriate enactment models that take in properties of the agent, such as exhaustion, and uses them to determine action outcomes (e.g., greater exhaustion can lead to greater defect density). We hope to address this fully in later work.

Our remaining criteria—modeling the product, work, rework, and quality—are explicitly built into the model as artifacts, problem reports/rework, and faults (unfixed problems), so we will show the

impacts of behavior on these simulation-wide outputs.

To demonstrate that the simulation satisfies our Agile-process-simulation criteria, we will simulate two processes: ad hoc and test-driven development (TDD). Ad hoc provides the control as it is a random selection of activities. In contrast, TDD is a well-defined incremental development practice with concrete intermediate goals (think, red, green, refactor) that must be satisfied in order to complete the increment. Moreover, it provides a good basis for a domain model as its constituent activities can be used to simulate other practices like generic test-first development (TFD) as well as test-last development (TLD). To make them comparable, the ad hoc process divides work into increments that require implemented and tested code in order to be declared complete. We will go into these in greater detail in later sections.

Along with the process definition (the scopes, contention resolution, and merge points), we want to show that we can model the impact of agent preferences on simulation outcomes. Agent preferences are captured using a utility function. In our experiment, utility motivates only process compliance. We defined two utility functions: one motivating agents to minimize the amount of work-in-progress (WiP) or minimize the number of increments that have been started but not completed, and another, constant-value function that causes the agent to select actions at random (all actions are equally "good", so the agent chooses at random).

To highlight a novel aspect of LAPSE, we will also show a multi-agent experiment where the agents followed different process using different utility functions. This will highlight the benefits of a shared process interface and domain model.

9.1 Domain Model

In order to simulate a process, we need a domain model. We do not seek an optimal domain model, nor do we seek something suited to more processes than TDD and the possible permutations thereof. Using the process specified in Section 5.1.1.2, we began with the most basic process, analyze and code [91], where analyze produces a set of requirements and code takes those requirements and produces the final product. However, rather than elaborating this as Royce does into waterfall [91], we elaborate the analysis into an incremental requirements generation and estimation procedure, producing sets of new user stories (roughly, requirements) for developers to work on. We then elaborate coding into TDD and integration of both tests and product code. We elaborate TDD in a similar manner to what we did earlier. The resultant domain model is depicted in Table 9.1.

Table 9.1: Experiment Domain Model - Inputs and Outputs

Activity	Input Artifacts	Output Artifacts
Define Requirements	Concept	Story
Define Increments	Story	Increment Requirement
Design Interfaces	Concept, Increment Requirement	Logical Interface Declaration
Declare Compilation Unit	Concept, Logical Interface Declaration, ¬ Product Compilation Unit	Product Compilation Unit
Declare Interface	Logical Interface Declaration, Product Compilation Unit, Logical Interface Declaration	Interface Declaration, Product Compilation Unit
Compile Product	Product Compilation Unit, Product Compilation Unit	Compiled Product
Plan Tests	Concept, Increment Requirement	Increment Test Plan
Declare Test Compilation Unit	Concept, Increment Test Plan, ¬ Test Compilation Unit	Test Compilation Unit
Define Logical Unit Test	Increment Requirement, Increment Test Plan, Logical Interface Declaration	Logical Unit Test
Define Concrete Unit Test	Logical Unit Test	Concrete Unit Test
Implement Test	Increment Requirement, Concrete Unit Test, Logical Interface Declaration, Test Compilation Unit	Implemented Test, Test Compilation Unit
Compile Tests	Test Compilation Unit, Compiled Product, Increment Requirement, Implemented Test, Logical Interface Declaration, Interface Declaration	Compiled Tests
Execute Tests (expect Failure)	Compiled Product, Compiled Tests, Increment Implementation, Increment Requirement	<METADATA>
Triage Unexpected Success	Compiled Product, Compiled Tests	<METADATA>
Implement Increment	Increment Requirement, Interface Declaration, Concept, Product Compilation Unit	Increment Implementation, Product Compilation Unit

Table 9.1 Continued: Experiment Domain Model - Inputs and Outputs

Activity	Input Artifacts	Output Artifacts
Execute Tests (expect Success)	Compiled Product, Compiled Tests, Increment Implementation, Increment Requirement	<METADATA>
Triage Unexpected Failure	Compiled Product, Compiled Tests	<METADATA>
Analyze Product Code Structure	Product Compilation Unit	<METADATA>
Analyze Test Code Structure	Test Compilation Unit	<METADATA>

In our earlier discussion of domain model properties, we highlighted a potential issue with domain models, specifically, that the expected result of some activities is based on the state of the world. For example, we expect tests to fail when the code under test is not yet written. To address this, we recommended using policies to direct the planner towards the expected output; however we do not use policies in our model. Instead, we flattened them out by defining sets of actions, each differentiated by distinct inputs, that the agent can take to arrive at the same behavior. Let us use testing as our example. When executing tests, an actor expects them to succeed if the code under test is present. If they are indeed successful, the actor can continue on to the next planned activity. However, if they fail, the actor triages the issue to determine the source of the unexpected failure, identifies the source of the problem, takes steps to correct it, and reruns the tests. Similarly, an actor may execute tests and expect failure, as might happen if the feature under test has not yet been implemented. If the tests then pass, the actor must figure out why (e.g., the feature already exists, there is a bug in the test code, or the environment is incorrectly set up). By flattening the policies into multiple versions of an activity distinguished by the activity's preconditions and also adding the activities necessary to respond to the unexpected, we are similarly able to generate plans.

In building the domain model, we identified a number of relationships among the artifacts. In order to determine the outcome of integration activities, we needed to model the resultant artifact as a container of other artifacts. Beyond simply showing the data has been integrated and preventing us from reintegrating the same artifact, this allowed us to, for example, determine if we should expect the tests to fail because the function under test is not present in the compilation unit. Additionally, we needed to model dependencies among artifacts. We focused on dependencies resulting from action executions, for example, a specific version of a compiled product that was used to compile the tests. This allowed us to

trigger recompilation of the tests when the dependency (the compiled product) changed.

9.2 Process Model

With a domain model in place, we can express the process model over it. There are three aspects of the process model that we must express: utility, work scoping, and coordination.

Utility We are interested in demonstrating that individual preference impacts simulation outcomes. In order to do this, we focused on two utility functions: minimize work in progress (MinWiP) and constant utility.

Minimize Work in Progress (MinWiP) Numerous processes emphasize maximizing value by eliminating waste, especially among Lean software development practices [82, 46]. One of the ways teams eliminate waste is by minimizing the amount of work-in-progress as partially completed work takes development effort away from higher value work leading to fewer things completed or poorer quality outputs [82]. If the team does return to the work, it may be unusable due to architecture, design, or code changes.

Unsurprisingly, a comparable focus appears in Agile literature. Jefferies' running tested features (RTF) metric motivates teams to complete features as soon as possible and deliver value [50], and Patton's "minimize output and maximize [both] outcome and impact" [79] pushes teams to know their limits and choose to do higher value work with high quality. Indeed, even the principles of the Agile manifesto claim that "our highest priority is to satisfy the customer through early and continuous delivery of valuable software" and that we want to "deliver working software frequently" [66]. Rapidly delivering value can be achieved by minimizing WiP because, by reducing waste and limiting the work to only what the team can manage, the team can focus on the work to be completed without directing effort towards other, potentially unnecessary work.

In order to express the desire to minimize WiP, we needed to define the work that the agent will minimize. We define a unit of work as a tested, integrated increment. For TDD, this means that the increment has both completed the Think, Red, Green, Refactor Test, and Refactor Product goals and has been integrated into the shared product and test suites. For the ad hoc process, it simply means that, for each increment, the product code exists and has been integrated with the code base and there is at least one executed and integrated test case exercising the increment.

Using the scopes to assist us in determining when a scope starts and ends, we compute the utility of minimizing work-in-progress (F_{minWiP}) as follows:

```
// Let:
// T be a tree containing all in-progress scopes
// P be the plan to assess, including expected world updates on completion of
// each action.
fun computeUtility(P): Double {
  val utility: Double = 0
  foreach( (A, W) an action and expected world in P ) {
    updateTree(T, W)

    val l = leavesInTree(T).length
    utility += 1.0 / ( l + 1.0 )
  }

  return utility / P.length
}
```

We realized that the amount of work-in-progress is equal to the number of active scopes and we wanted to ensure the utility value is on $[0, 1]$ where 1 is more desirable than 0. Thus, we wanted the inverse of the amount of work-in-progress. To prevent us from dividing by zero, we add 1 to the denominator. Because we are assessing an entire plan, we want to find the amount of in-progress work for the entire plan. Thus, we average the individual action utility to compute the utility for the plan.

Constant-value To contrast the effects of minimizing WiP, we wanted a control; some utility that would not influence an agent's behavior. We achieve this using a constant-value utility. When all actions, and, therefore, all candidate plans are equally beneficial, the agent breaks the tie by selecting a plan at random.

While we discussed minimize WiP and constant-value as two utility functions on their own, we found that these were insufficient to motivate agents. Agents required additional motivation to mimic human behavior, such as choosing not to idle when there is work to be done and not to repeat actions. Moreover, we wanted to prevent agents from blocking each other and ensure that we merge work as soon as possible. So, in addition to the aforementioned process-preference aspects of utility, we included inducements for agents not to idle and to exit when possible, to not repeat actions, and to merge completed work as soon as possible.

To induce the agent not to repeat actions ($I_{\neg\text{repeat}}$), we track where the agent has been and return 1 if the intersection between the plan and the set of completed actions is non-empty or 0 otherwise. For the latter two inducements, we want to motivate agents to perform desirable actions (exit and pre-specified actions) as soon as possible and undesirable actions (idle) as late as possible, so we use the following

linear decay function:

$$I = \sum_{k=0}^{p-1}(-2p^{-2}k - p^{-2} + 2p^{-1})s_k$$

where p is the length of the plan and s_k is the score of the action at k. For biasing towards exiting and against idling ($I_{\neg\text{idle}}$), s_k is 1 if the action at k is exit, 0 if it is idle, and 0.5 otherwise. For biasing towards merge actions (or any pre-specified action; I_{merge}), s_k is 1 if the action at k is in the set of specified actions or 0 otherwise.

We average our previously defined utility functions together to get the new utility functions—minimize work-in-progress (U_{minWiP}) and constant-value (U_{const})—we will use within the experiments:

$$U_{\text{minWiP}} = \frac{F_{\text{minWiP}} + I_{\neg\text{idle}} + I_{\neg\text{repeat}} + I_{\text{merge}}}{4}$$

$$U_{\text{const}} = \frac{0 + I_{\neg\text{idle}} + I_{\neg\text{repeat}} + I_{\text{merge}}}{3}$$

Work Scoping In order to compute the utility, we must define a measurable unit of work. Initially, we expressed work in terms of the goal state the agent directs its effort toward. Simply focusing on the goal made planning easy as we could focus on selecting actions that would contribute to satisfying the goal. However, we had a difficult time determining which activities and intermediate artifacts belonged to which unit of work and how much time was spent completing a work item. In traditional project management, the bounds of work, including the specific activities to be performed, are described in the work breakdown structure. This establishes a scope for the work—a set of bounds around the effort indicating both when the work has started/ended as well as which activities contribute to goal achievement. Similarly, many Agile processes teams record when work on a feature begins and ends in order to form metrics. However, within an Agile process, it is unclear what the scope of the work is since the activities required to complete it are not pre-prescribed. Instead, developers are free to determine what actions to take to achieve the goal. This makes the work scope unclear as the activities of multiple features could be interleaved or could be different from those performed by others on the team. Further, the lack of clarity of the work scope makes it particularly difficult to perform some utility computations, like minimize work in progress.

Rather than prescribing the activities belonging to each scope, we determine work scope dynamically by pairing each goal with a criterion to indicate when the agent has begun work towards the goal. With the start and end point clearly defined, we need only track the scope of the activities between those points. Since activities from different scopes may be interleaved—as might occur if the work for one scope is blocked—we need a way to track which activities belong within each scope. Because we have an artifact/data-centric simulation model, we do not track the scope of activities, but instead track the

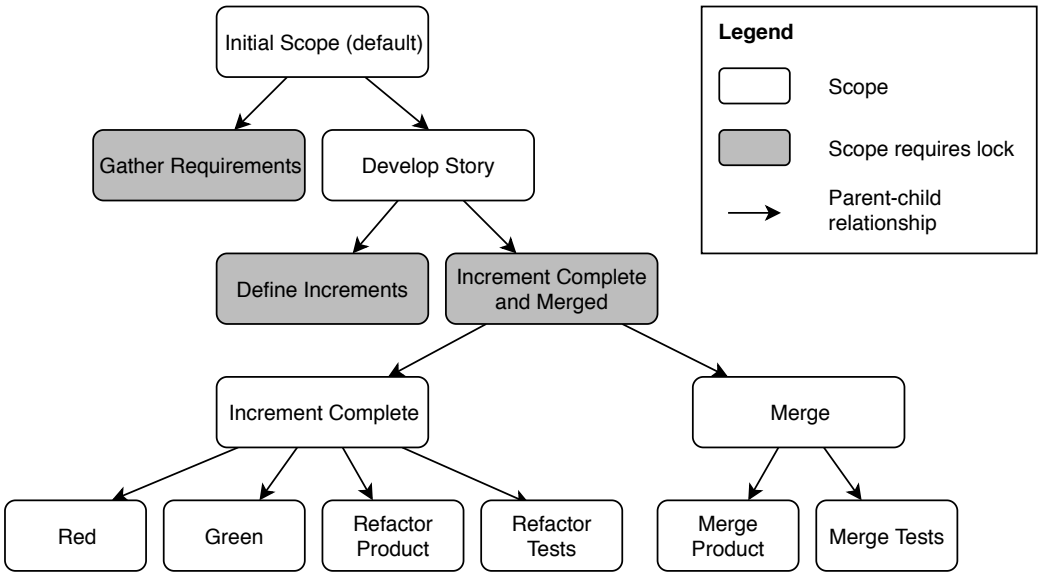

Figure 9.1: The TDD scope hierarchy used in our experiment. In our experiment, we omitted the 'Develop Story' scope as we only modeled a single story with one or more increments. The think scope became 'Define Interfaces'.

scope of the artifacts produced: each activity has an input artifact whose scope is inherited by (passed on to) the activity's outputs. This scope inheritance is defined as part of the activity. When we start a new scope, we override the scope of the outputs and treat the new scope as a child (subscope) of the original scope. By tracking the scope of each artifact as well as its parentage, we can find all artifacts belonging to a scope without prescribing the activities an agent can perform. Further, we can divide work into smaller and smaller units and reason over subscopes. For example, we can mark a feature as complete under TDD when all of its increments are complete, and we can mark an increment complete when its tests pass and both the product code and the tests have been refactored.

The scope hierarchy for TDD as used in our experiment is shown in Figure 9.1. Under this process specification, the work for an increment is complete when it is developed, tested, and refactored (`Increment Complete`); integrated (`Merged`) with the existing product features; and accepted by the source of truth (part of `Increment Complete and Merged`). The AdHoc process specification is similar to that of TDD. Many of the scopes are shared. However, we replace the child scopes of `Increment Complete` with the more generic `Implemented` and `Tested`, thereby omitting refactoring from our completion criteria. Refactoring may still be performed by an agent, but it is not required for work completion.

Coordination Upon completion of a work scope, what should an agent do with the completed work? How does work get integrated? The process model specifies both when to share beliefs with others and how to merge artifacts when the local copy differs from the source of truth. In LAPSE, when an increment is complete, the agent attempts to publish all artifacts belonging to the increment to the source of truth. The source of truth analyzes the artifacts for conflicts (existing artifacts that subsume other artifacts), and, if none are found, the source of truth accepts the changes. If, on the other hand, there is a merge conflict, the source of truth rejects the changes and notifies the agent that the change needs merging. The agent performs the merge and attempts to publish the updated artifacts. From our domain model, we need to support merging both the product and test compilation units.

To reduce the likelihood of duplicate effort and minimize potential conflicts, agents in LAPSE both claim the work they will perform and honor the claims of others; effectively locking artifacts at fixed points in the process. If all work is claimed by other agents, leaving nothing for an agent to do, the agent idles. As soon as completed work is published to the source of truth, any agent can build on the published work (including fixing problems) by acquiring a new claim to the artifacts. The work claim/lock points in our simulation model are illustrated in Figure 9.1.

9.3 Activity Enactment

In this work we aim to show that we can simulate Agile processes with a particular focus on the cognitive aspects of a multi-agent simulation. Consequently, we wish to demonstrate the agent's ability to respond to unexpected activity outputs. By modeling both the actual outputs of each activity and the time required to complete an activity we can both demonstrate the agent's responsiveness to change and derive effort and duration measures.

Recall that, during planning, we use the idealized form of each activity as actors cannot correctly forecast the activity's output, such as the expected quality of their work. We embrace this uncertainty at plan time and resolve it during enactment, where we have as much knowledge as we can get about the agent and the state of the world. During enactment, we model the expected results including the faults injected by the agent as well as the specific output artifacts and quantities thereof. For our experiment, most activities emit the idealized output used in planning, but for some we return higher quantities of the artifact (e.g., breaking down a story into multiple increments) or we inject faults that will be found later in the process. Table 9.2 describes the specific, non-ideal enactment behaviors.

Table 9.2: Activities with non-ideal enactment behaviors.

Activity	Behavior (First run only; not rework/defect triggered)
Define Increments	Emits a pre-defined number of increments for the story.
Implement Test	Adds 2 faults (of different types) to the test artifact.
Analyze Test Code Structure	Finds faults that can be resolved through refactoring and generates problem reports for them.
Compile Tests	Finds and reports all syntax errors. If none are found, generates the compiled test suite.

The second part of enactment, modeling the time to complete activities, is important for accurate results. Because we are only demonstrating the ability to model Agile processes, we use a constant-time model for each activity. In follow-up work, we expect to develop activity-time models for our simulation.

Chapter 10

Results and Discussion

Having constructed the Lean/Agile Process Simulation Environment (LAPSE), our reference Agile process simulator, we want to show that it will satisfy our requirements and evaluate processes in terms of their expected outputs, namely the expected project cost, schedule, scope (function), and quality. While it is outside the scope of this work to produce accurate estimates, we will demonstrate that, with the right model, we can compute these expected outputs.

Focusing on simulating a single-story product, we developed a number of different scenarios to highlight our simulation. We began with a scenario in which the planner accurately predicts the activity enactment outcome for all activities (perfect planning or the ideal case). Building on the ideal scenario, we overrode the ideal enactments (see Table 9.2) and gradually increased both the number of increments being developed and the number of agents in the simulation. For each of these runs, all agents in the simulation had the same utility function, but what would happen if we had a team where the processes varied? We simulated that as well, using the common join points between our two processes (e.g., a shared definition of done) to allow work to be merged. The results of running each scenario, averaged over ten runs each, are shown in Table 10.1.

In the table, we show the expected total effort (the labor aspect of project cost), the simulation time (project duration), the number of completed increments (project scope), and defect density (project quality; the average number of unfixed faults per increment). With additional metadata generated for each increment, we could quantify the amount of completed function using some form of point system. Thus, LAPSE satisfies our required assessment dimensions (Section 2.3).

Individual Behavior and Process Compliance Within LAPSE, we capture individual preference through utility functions. Using two distinct utility functions, we highlight not only that agents can

Table 10.1: The average results over 10 simulation runs for each set of inputs (process, utility, number of agents, and number of increments per story).

Experiment Set	**Ideal**	**Uniform Agents**								**Nonuniform Agents**
Process	TDD	AdHoc	TDD	AdHoc	AdHoc	TDD	AdHoc	TDD	TDD	AdHoc/TDD
Utility	MinWiP	Const	MinWiP	Const	MinWiP	MinWiP	Const	MinWiP	MinWiP	Const/ MinWiP
Number of Agents	1	1	1	2	2	2	2	2	3	2
Increments	1	2	2	2	2	2	4	4	5	4
Run Time (secs)	0.098	2.116	1.613	1.609	1.593	1.553	36.907	26.000	321.083	24.665
Simulation Time (ticks)	18	39	48.9	33.2	31.6	33.8	52.5	65.3	71.5	64.3
Increments Completed	1	2	2	2	2	2	4	4	5	4
Latent Product Faults	0	1	1	1	1	1	1	1	1	1
Density (faults per increment)	0	0.5	0.5	0.5	0.5	0.5	0.25	0.25	0.2	0.25
Problems Reported	0	4	4	3.9	3.8	4.0	7.1	8	10	7.9
Resolved Problem Reports	0	4	4	3.9	3.8	4.0	7.1	8	10	7.9
Avg Fault Detection Time (ticks)	N/A	4.0	4.8	2.6	1.7	5.1	3.3	4.8	4.4	4.3
Avg Problem Resolution Time (ticks)	N/A	2.6	1.0	2.2	1.4	1.0	3.2	1.0	1.0	3.4
Total Effort (ticks)	18	39	48.9	53.8	54.5	59.8	86.9	115.0	156.8	101.2
Avg Effort (ticks)	18	39	48.9	26.9	27.3	29.9	43.5	57.5	52.3	50.6
Avg Utilization	1	1	1	0.814	0.862	0.884	0.835	0.890	0.739	0.788

compose complete process executions—process-compliant activity sequences—without full prescription, but also that the constructed executions comply with the agent's preferences. Listing 10.1 shows a complete process execution from our ideal simulation runs (TDD-compliant with the minWiP utility), demonstrating our agents' ability to compose process-compliant and preference-compliant executions on-the-fly despite having a planning horizon of 3 actions. Building on this, we executed non-ideal simulations to show that agents can recover from the unexpected and still produce process-compliant and preference-compliant executions. Listing 10.2 shows two such process executions for the AdHoc process. The first (Listing 10.2a) uses the constant-value utility function and the second (Listing 10.2b) uses the MinWiP utility function. Observe that when using the MinWiP utility, the agent completes scopes (phases and increments) before starting another one at the same nesting depth. This is the expected behavior as starting a nested scope keeps the amount of work in progress constant. In contrast, with the constant-value utility the agent interleaves work on multiple increments and does not produce a clear distinction between implementation and testing phases. With just this example of the behavior captured in LAPSE, we can readily observe that preference, expressed as utility, greatly impacts the agent's entire process execution.

Listing 10.1: Single Agent Execution of TDD with Ideal Outcomes—simplified output numbered by simulation time

```
0: SCOPE_START(name="Gather Requirements", params=[ "concept_00" ], parent="DEFAULT SCOPE_00")
0: Define Requirements("concept_00", newScope="Gather Requirements_00")
1: SCOPE_END(name="Gather Requirements", start="Gather Requirements_00", params=[ "story_01" ])
1: SCOPE_START(name="Define Increments for Story", params=[ "story_01" ],
             parent="DEFAULT SCOPE_00")
1: Define Increments("story_01",
                  newScope="Define Increments for Story_01")
2: SCOPE_END(name="Define Increments for Story",
           start="Define Increments for Story_01"
           params=[ "story_01" ])
2: SCOPE_START(name="Increment: Complete And Merged",
             params=[ "incrementDefinition_02" ],
             parent="DEFAULT SCOPE_00")
2: SCOPE_START(name="Increment: Complete",
             params=[ "incrementDefinition_02", "concept_00" ],
             newScope="Increment: Complete And Merged_02")
2: Design Interface(s)("incrementDefinition_02", "concept_00",
                    newScope="Increment: Complete_02")
```

```
 3: Declare Compilation Unit("logicalInterfaceDeclaration_03", "concept_00")
 4: Declare Interface("logicalInterfaceDeclaration_03", "productCompilationUnit_00",
                   "logicalInterfaceDeclaration_03")
 5: SCOPE_START(name="Increment: Red",
             params=[ "incrementDefinition_02", "concept_00" ],
             parent="Increment: Complete_02")
 5: Plan Tests("incrementDefinition_02", "concept_00",
            newScope="Increment: Red_02")
 6: Define Logical Unit Test(s)("incrementTestPlan_02", "incrementDefinition_02",
                           "logicalInterfaceDeclaration_03")
 7: Declare Test Compilation Unit("incrementTestPlan_02", "concept_00")
 8: Define Concrete Unit Test(s)("logicalUnitTest_04")
 9: Implement Test("incrementDefinition_02", "concreteUnitTest_05", "testCompilationUnit_00")
10: Compile Product("productCompilationUnit_00", "interfaceDeclaration_03")
11: Compile Tests("testCompilationUnit_00", "compiledProduct_00", "incrementDefinition_02", "
     implementedTest_05")
12: Execute Tests (Expecting Failure)("compiledProduct_00", "compiledTests_00", "
     incrementDefinition_02")
13: SCOPE_END(name="Increment: Red",
            start="Increment: Red_02",
            params=[ "compiledProduct_00", "compiledTests_00", "implementedTest_05" ])
13: SCOPE_START(name="Increment: Green",
              params=[ "incrementDefinition_02", "concept_00", "productCompilationUnit_00", "
                   implementedTest_05" ],
              parent="Increment: Complete_02")
13: Implement Increment("incrementDefinition_02", "interfaceDeclaration_03", "concept_00", "
     productCompilationUnit_00", "interfaceDeclaration_03",
                     newScope="Increment: Green_02")
14: Compile Product("productCompilationUnit_00", "incrementImplementation_02", "
     interfaceDeclaration_03")
15: Execute Tests (Expecting Success)("compiledProduct_00", "compiledTests_00", "
     incrementDefinition_02")
16: SCOPE_END(name="Increment: Green",
            start="Increment: Green_02",
            params=[ "compiledProduct_00", "compiledTests_00", "incrementImplementation_02" ])
16: SCOPE_START(name="Increment: Refactor Product",
              params=[ "productCompilationUnit_00", "incrementImplementation_02", "
                   compiledProduct_00", "compiledTests_00",) ],
              parent="Increment: Complete_02")
16: Analyze Product Code Structure("productCompilationUnit_00",
```

```
                                newScope="Increment: Refactor Product_02")
17: SCOPE_END(name="Increment: Refactor Product",
             start="Increment: Refactor Product_02",
             params=[ "productCompilationUnit_00", "compiledProduct_00", "compiledTests_00", "
                  incrementDefinition_02" ])
17: SCOPE_START(name="Increment: Refactor Tests",
               params=[ "testCompilationUnit_00", "implementedTest_05", "compiledTests_00", "
                    compiledProduct_00", "incrementDefinition_02" ],
               parent="Increment: Complete_02")
17: Analyze Test Code Structure("testCompilationUnit_00",
                             newScope="Increment: Refactor Tests_02")
18: SCOPE_END(name="Increment: Refactor Tests",
             start="Increment: Refactor Tests_02",
             params=[ "testCompilationUnit_00", "implementedTest_05", "compiledTests_00", "
                  compiledProduct_00", "incrementDefinition_02" ])
18: SCOPE_END(name="Increment: Complete",
             start="Increment: Complete_02",
             completedSubscopes=[ "Increment: Red_02", "Increment: Green_02", "Increment:
                  Refactor Product_02", "Increment: Refactor Tests_02" ],
             params=[ "compiledProduct_00", "compiledTests_00", "incrementDefinition_02", "
                  productCompilationUnit_00", "testCompilationUnit_00" ])
18: SCOPE_END(name="Increment: Complete And Merged",
             start="Increment: Complete And Merged_02",
             completedSubscopes=[ "Increment: Complete_02" ],
             params=[])
END
```

Communication and Coordination Within our simulation, we modeled interactions with the source of truth (SoT) as communication (illustrated in Listing 10.3). The SoT supported two types of messages: lock requests (work claims) and cognitive model/world updates. For lock requests, the SoT assigns the lock if it has not already been assigned and responds with the owner of the requested lock (lines 1-4, 7-10, 13-20). With model/world updates, the SoT checks for conflicting changes and either responds with a "merge required" message to the sender (line 45 – we do not show the cause of this response) or it updates itself and forwards the changes to the other agents (lines 5-6, 11-12, 21-32, 46-57). In this way, the SoT is the mediator between agents akin to a communication channel with messages, containing beliefs/knowledge, flowing through the SoT out to other agents. Because the SoT is just another agent within the simulation, we expect that we can expand the communication support to enable unmediated communication and knowledge-sharing among an arbitrary set of agents.

Rework/Responding to change In LAPSE, rework can be implicitly or explicitly triggered. Problem reports are explicit triggers of rework. If a problem report identifies a fault (problem) in an artifact produced by an activity, then that activity is rerun to address/fix the problem and a new artifact is produced. Implicitly triggered rework is modeled by tracking the creation time or version of each object. If an activity's inputs are updated (they are newer than the version previously used to build its outputs), the activity is rerun. This generates new versions of its outputs. All rework, can trigger cascading updates—implicitly triggered rework in downstream activities causing them to run again. Listing 10.2a, lines 41-45 shows this. Here, the analysis identifies structural issues in the test suite requiring refactoring. Once refactored (shown here as re-implementation), the reworked test implementation artifacts trigger recompilation and another analysis.

Using this rework mechanism, new work can be modeled by introducing a new artifact or problem report to the world (such as a new requirement/story). However, modeling work roll-back is more challenging. We did not attempt it in our experiments as had no roll-back triggering actions in our domain model. To model work-rollback, we expect to model it as new work (a new artifact or problem report) that mimics removing existing artifacts. This allows us to track wasted effort within the simulation and keeps us from violating our assumption that nothing is removed from the world.

Listing 10.2: Process execution fragments (sequences of activities) with scope start/end markers for agents following the AdHoc process. Each agent adhered to a different utility function.

(a) Constant-value utility function

```
...
START(Define Story Increments)
Define Increments
END(Define Story Increments)
...
START(Increment: Complete And Merged)
START(Increment: Complete)
START(Increment: Implementation)
Design Interface
START(Increment: Complete And Merged)
START(Increment: Complete)
START(Increment: Testing)
Plan Tests
START(Increment: Implementation)
Design Interface
Declare Test Compilation Unit
Declare Compilation Unit
Analyze Product Code Structure
Analyze Test Code Structure
START(Increment: Testing)
Plan Tests
Define Logical Unit Test
Declare Interface
Analyze Product Code Structure
Define Logical Unit Test
Define Concrete Unit Test
Declare Interface
Implement Increment
Define Concrete Unit Test
Compile Product
END(Increment: Implementation)
Implement Increment
Implement Test
Analyze Test Code Structure
Compile Product
END(Increment: Implementation)
Analyze Product Code Structure
Compile Tests
Implement Test
Compile Tests
Analyze Test Code Structure
Implement Test
Implement Test
Compile Tests
Analyze Test Code Structure
Execute Tests
END(Increment: Testing)
END(Increment: Testing)
END(Increment: Complete)
END(Increment: Complete)
END(Increment: Complete And Merged)
END(Increment: Complete And Merged)
```

(b) MinWiP utility function

```
...
START(Gather Requirements)
Define Requirements
END(Gather Requirements)
...
START(Increment: Complete And Merged)
START(Increment: Complete)
START(Increment: Implementation)
Design Interface
Declare Compilation Unit
Analyze Product Code Structure
Declare Interface
Analyze Product Code Structure
Compile Product
Implement Increment
Analyze Product Code Structure
Compile Product
END(Increment: Implementation)
START(Increment: Testing)
Plan Tests
Define Logical Unit Test
Define Concrete Unit Test
Declare Test Compilation Unit
Analyze Test Code Structure
Implement Test
Compile Tests
Analyze Test Code Structure
Implement Test
Compile Tests
Execute Tests
END(Increment: Testing)
END(Increment: Complete)
START(Merge)
START(Merge Test Compilation Unit)
Merge Test Compilation Unit
START(Merge Product Compilation Unit)
Merge Product Compilation Unit
Compile Product
Execute Tests
END(Merge Product Compilation Unit)
Compile Tests
Execute Tests
END(Merge Test Compilation Unit)
END(Merge)
END(Increment: Complete And Merged)
```

Listing 10.3: Communication with the Source of Truth Agent (2 Agents; TDD; minWIP).

```
Received from A1: LockRequest("concept_00")
Sent to A1: Lock(A1, "concept_00")
Received from A0: LockRequest("concept_00")
Sent to A0: Lock(A1, "concept_00")
Received from A1 and sent to ALL:
    Artifact("story_01")
Received from A1: LockRequest("story_01")
Sent to A1: Lock(A1, "story_01")
Received from A0: LockRequest("story_01")
Sent to A0: Lock(A1, "story_01")
Received from A1 AND sent to ALL:
    Artifact("incrementDefinition_02"), Artifact("incrementDefinition_03")
Received from A1: LockRequest("incrementDefinition_03")
Sent to A1: Lock(A1, "incrementDefinition_03")
Received from A0: LockRequest("incrementDefinition_03")
Sent to A0: Lock(A1, "incrementDefinition_03")
Received from A0: LockRequest("incrementDefinition_02")
Sent to A0: Lock(A0, "incrementDefinition_02")
Received from A1: LockRequest("incrementDefinition_02")
Sent to A1: Lock(A0, "incrementDefinition_02")
Received from A0 AND sent to ALL:
    Artifact("logicalUnitTest_04"), Artifact("incrementTestPlan_02"),
    Artifact("concreteUnitTest_05"), Artifact("implementedTest_05"),
    Artifact("productCompilationUnit_00"), Artifact("compiledTests_00"),
    Artifact("implementedTest_05"), Artifact("implementedTest_05"),
    Artifact("interfaceDeclaration_06"), Artifact("logicalInterfaceDeclaration_06"),
    Artifact("testCompilationUnit_00"), Artifact("incrementImplementation_02"),
    Artifact("compiledProduct_00"),
    ProblemReport("refactorableFaultType", "implementedTest_05"),
    ProblemReport("syntaxFaultType", "implementedTest_05"),
    ProblemReportResolved("syntaxFaultType", "implementedTest_05"),
    ProblemReportResolved("refactorableFaultType", "implementedTest_05")
Received from A1:
    Artifact("incrementImplementation_03"), Artifact("productCompilationUnit_00"),
    Artifact("testCompilationUnit_00"), Artifact("implementedTest_07"),
    Artifact("implementedTest_07"), Artifact("concreteUnitTest_07"),
    Artifact("incrementTestPlan_03"), Artifact("logicalInterfaceDeclaration_08"),
    Artifact("compiledProduct_00"), Artifact("implementedTest_07"),
    Artifact("compiledTests_00"), Artifact("interfaceDeclaration_08"),
    Artifact("logicalUnitTest_09"),
    ProblemReport("refactorableFaultType", "implementedTest_07"),
    ProblemReport("syntaxFaultType", "implementedTest_07"),
    ProblemReportResolved("syntaxFaultType", "implementedTest_07"),
    ProblemReportResolved("refactorableFaultType", "implementedTest_07")
Sent to A1: UpdateRejected("merge required")
Received from A1 AND sent to ALL:
    Artifact("incrementImplementation_03"), Artifact("implementedTest_07"),
    Artifact("compiledProduct_00"), Artifact("productCompilationUnit_00"),
    Artifact("testCompilationUnit_00"), Artifact("logicalInterfaceDeclaration_08"),
    Artifact("implementedTest_07"), Artifact("implementedTest_07"),
    Artifact("concreteUnitTest_07"), Artifact("incrementTestPlan_03"),
    Artifact("interfaceDeclaration_08"), Artifact("compiledTests_00"),
    Artifact("logicalUnitTest_09"),
    ProblemReport("refactorableFaultType", "implementedTest_07"),
    ProblemReport("syntaxFaultType", "implementedTest_07"),
    ProblemReportResolved("syntaxFaultType", "implementedTest_07"),
    ProblemReportResolved("refactorableFaultType", "implementedTest_07")
```

10.1 Threats to Validity

As with any work, there is the potential to inject faults or otherwise bias the results. Runeson, Host, Rainer, and Regnell provides a categorization of potential threats to validity: construct validity, internal validity, external validity, and reliability [93]; we will address each of these.

Construct Validity In this work, we constructed a simulation to model Agile processes, specifically focusing on the behavioral aspects of the individual enacting the process activities. We chose to use simulation to compare processes based on simulation's suitability to the task. However, simulation is contraindicated in some situations; when the model is ill-suited to answering the types of questions posed or can be done more efficiently through direct experimentation, is infeasible due to complexity or lack of data, is unverifiable, is cost-ineffective, or is insufficiently resourced [8]. We addressed suitability thoroughly earlier in this work, while cost and resource concerns are no longer pertinent. In addressing verifiability, this work has attempted to break the problem down into smaller pieces and verify them. We decomposed the problem using both the reference model and agent eval-loop to guide this process. Focusing on the cognitive aspects of simulation, we showed how to construct a domain model that will facilitate planning and built on its properties within the simulation. We left some parts of the simulation, like enactment models, to future work. Upon completion of those parts, we can verify the integration of all of the pieces. However, even using stand-in/stubbed components, we were able to validate—by changing inputs and observing expected changes in outputs—that our simulation is both feasible and satisfies our requirements.

Were our requirements wrong or flawed in some way? We attempted to minimize this by directly mapping from the values and principles of the Agile Manifesto [66] to specific properties and justified each of these mappings. Because we map each of the values and principles to at least one property (Table 2.1) we expect that the requirements are complete with respect to the Manifesto. From these properties, we constructed our requirements, ensuring that each property mapped to one or more requirements

Internal Validity and Reliability Confounding, the misattribution of outcome changes to variations in one input variable instead of another, was a major concern for us during simulation as processes and utility functions are pretty closely tied. We attempted to minimize the potential for confounding by metamorphosing only one input at a time and ensuring it independently changed the results in expected ways. We did this to show the impacts of utility and process on project outcomes (for example, the AdHoc+Const, AdHoc+MinWiP, and TDD+MinWiP experiments with 2 agents; Table 2.1). We also ran each simulation ten times with different seeds for the random number generator to ensure results were consistent across runs.

External Validity Throughout this work, we focus heavily on TDD methods, yet other Agile processes and practices exist. To show that our approach works with non-TDD approaches, we both showed our initial domain model, generated to show the iterative elaboration technique (Section 5.1.3), can replicate other practices like test-last development, and we demonstrated our technique over an Ad Hoc iterative "process". Further, we constructed our model requirements against the commonly accepted definition of Agile, the Agile Manifesto [66], so by satisfying these requirements we expect that we can model an arbitrary Agile process.

In previous chapters we described how to model the product and quality. Building on that, we demonstrated in this chapter that LAPSE can

1. construct a process execution in the absence of a fully-ordered process specification,
2. can model self-determination, agent communication, and rework, and
3. produce cost, schedule, quality, and scope metrics/measures for process evaluation.

Therefore, we can reasonably conclude that it is not only possible to create a simulation model that both satisfies our requirements and can model Agile processes, but that LAPSE accomplishes this.

Chapter 11

Conclusion

Process selection and adoption is risky as selecting an ill-fitting process can hinder project success. To reduce this risk, we want to simulate candidate processes, particularly Agile processes, prior to adoption. Because of Agile's individual focus and the real impacts people have on project success, we expect that a high-fidelity Agile process simulation, one that models individual decision-making and the resultant behavior, will produce more accurate predictions and therefore provide better decision support than process selection and tailoring guides, pilot projects, and existing process simulations. In order to compare existing techniques/simulations, we extracted properties of Agile processes from the Agile Manifesto [66] and from them derived requirements for an Agile process simulation suited to comparing processes with respect to their project outcomes. After reviewing existing simulations we found none that satisfied our simulation criteria. That left us with more questions; particularly:

1. Is it possible to create a simulation model that satisfies our Agile properties (are our criteria satisfiable)?
2. If so, what are the constructs (components) that might be needed in the simulation? Most importantly:
    a. How can we model lightweight processes (processes that do not provide complete prescription) such as Agile processes?
    b. How can we model the impact of each individual contributor on the project?

To answer these questions, we set out to construct a simulation that would satisfy our criteria and allow us to simulate Agile processes. Because we want to model individual behavior and its impacts on project outcomes, we chose agent-based simulation as agents are natural proxies for humans. We began by defining a reference model that captures the base constructs and relationships we wish to model

within the simulation. Because Agile processes are lightweight and declarative in nature, we needed a model that permits flexibility by underspecification and supports an agent-based simulation.

We then defined the different aspects of the simulation, starting with a single agent and scaling up. To model agent behavior, we took a goal-oriented approach. Here, we required two primitives, actions and goals. Actions, including development activities, could be chained together into plans according to their inherent data constraints. Plans are then assessed using modeler-defined utility function that scores plans according to the relative desirability of the plan for the agent. Plans that contribute to goal satisfaction, for example, would be more desirable and would get a higher utility value than plans that do not. The selected plan would then be enacted by the agent and the world updated. However, it is unclear what activities are required to simulate Agile processes and how they fit together. Others authors have attempted to construct new processes from process fragments, but they are concerned with preserving control flow, which may limit the behaviors of the agent. Worse still, imperative models predefine both the specific sequence of activities at the outset of the simulation, prescribing specifically how to reach the goal. In contrast, Agile processes describe the what needs to be done—goals for the actor—and allow the actor to determine how best to achieve them. This approach to processes lends itself to a declarative model. However, there is no existing set of declaratively defined activities for Agile processes, nor is there guidance for creating one. Without a set of actions over which to reason, an agent cannot figure out how to reach the goal. To address this, we developed an algorithm for interactively elaborating the set of activities, called the planning domain model, and inductively showed that each elaboration preserved both model validity (internal completeness and internal consistency) and replannability, attributes essential to planning.

Having created a single agent, we focused on the simulation driver that manages simulation progression, the shared world, and communication channels. We complemented the domain model with actions to aid in sharing beliefs among agents and resolving contention. Further, we specified the process as a hierarchy of scopes, boundaries around a unit of effort directed towards satisfying a goal, each ultimately composed of a specific activity execution. Because we demarcate the scope start and stop with specific beliefs and because we express scope completion (goal satisfaction) in terms of beliefs, we can express control-flow constraints over the set of scopes.

We set out to evaluate candidate Agile processes with respect to their expected output, so we must ask, is it possible to create a process evaluation framework that can satisfy our Agile properties, and, therefore, model Agile processes? Yes. We constructed the Lean/Agile Process Simulation Environment (LAPSE), a reference simulator, to demonstrate this. Specifically, we constructed a domain model, targeting TDD, and used it, along with two utility functions and two process definitions (scope hierarchies and synchronization points), to simulate people following the prescribed process. To show the independent

impacts of the process and preference (utility function) on outcomes, we varied each of these separately and observed the expected changes in the output (a metamorphic relation). We further showed that LAPSE simulates processes without full prescription, models the effects of rework, supports communication among agents, and generates the desired metrics and measures.

Contributions This work is an important step towards greater decision support to process engineers, developers, and project managers adopting Agile processes. As such, our primary contributions are:

- Criteria, rooted in the Agile Manifesto, for determining if a simulation can model Agile processes
- Identification of the constructs needed to construct a simulation, with a particular focus on those required for modeling individual decision-making
  - An algorithm for constructing a planning domain model that ensures plan validity
- LAPSE, our declarative, agent-based reference simulator, that shows both that our criteria are satisfiable and demonstrates how the constructs fit together

Future Directions

Extensions/Enhancements Ultimately, we want to determine if a high-fidelity simulator, such as LAPSE, can provide more accurate outcome predictions than those evaluation frameworks that do not explicitly model the individual. This work focuses on modeling process execution assembly (process-compliant activity selection and sequencing) for lightweight/Agile processes. Building on this foundation, LAPSE should be extended, by incorporating activity enactment models, to accurately reflect specific action outcomes and more systematically account for fault/problem injections. Such models would need to classify the types of problems introduced into the product and under what conditions they are found. They would also address the effects of problems in tests on product quality and development effort as well as the reverse. Additionally, we need time models to predict the activity duration within the simulation (the bidding time). Together, these will allow LAPSE to generate realistic cost, schedule, scope, and quality predictions. To complement these models, we can characterize the human factors/personal traits of the simulated actors and integrate them into utility functions and enactment models (e.g., by integrating and building on TEAKS' work performance model [71]).

In naming our simulation LAPSE, we highlight one of the aspirations of this work: to support any lightweight process, including Lean. While some people classify Lean as Agile, we still want to properly assess and ensure that LAPSE satisfies the different properties of Lean processes. Moreover, because traditional development models require less flexibility, we suspect, but must still confirm, that LAPSE

can also model traditional approaches, allowing for greater comparisons and the modeling of hybrid process models.

One of the base assumptions of this work is that the process selection is for a greenfield project. However, there are relatively few greenfield projects compared to the number of projects that are looking to improve their process. By providing initial state information and a robust model of individual behavioral traits as inputs, we expect that LAPSE could support process improvement decisions.

Beyond the scope of decision support, this work could aid in the education of software engineers and project managers. By providing a playground in which to explore, learners can observe the effects of their choices on team outcomes. However, this does not need to be limited to experimenting with different inputs and observing the outputs. By treating the learner as an agent within the simulation, LAPSE could be adapted to allow live interactions and virtually embed the learner in the team he is modeling.

Analyses From tacit observation, developers seem to find things that work for them, patterns of behavior, and stick to them. Thus, over time, it seems likely that a software engineer will develop an affinity to a certain sequence of actions. The current simulation does not account for these affinities or the consequence of deviation on individual productivity and morale. From process change models, like the Satir model, we see that there can be a negative impact from change and we suspect that deviating from some set of these affinities will hinder individual and team output [12, 14]. We'd like to set up LAPSE to aid us in this analysis.

While we focused on decision support, simulations have also been used to explain a phenomena. By developing theories and testing them within a simulation, researchers are able to quickly validate the theory. Within the software engineering discipline, there has been a lot of interest in pair programming (see [9]). Regarding the impacts of pair programming, there are a number of conflicting studies [33, 113]. We expect that LAPSE's ability to predict outcomes will enable us to explore theories explaining why this technique may work for some and not for others and allow us to experiment with different causal factors (e.g., motivation, personality, preference, and expertise) expressible as utility.

References

[1] 2018. 12th Annual State of Agile Report. Technical report. Version One, (April 2018). Retrieved 02/28/2019 from `https://explore.versionone.com/state-of-agile/versionone-12th-annual-state-of-agile-report`.

[2] 2017. *A Guide to the Project Management Body of Knowledge (PMBOK® Guide).* (6th edition). Project Management Institute, Incorporated, Newtown Square, PA.

[3] Lars Ackermann and Stefan Schönig. 2016. MuDePS: Multi-perspective Declarative Process Simulation. In *BPM (Demos)*, 12–16.

[4] Lars Ackermann, Stefan Schönig, and Stefan Jablonski. 2016. Simulation of multi-perspective declarative process models. In *International Conference on Business Process Management*. Springer, 61–73.

[5] Allan J. Albrecht and John E. Gaffney Jr. 1983. Software Function, Source Lines of Code, and Development Effort Prediction: a Software Science Validation - IEEE Journals & Magazine. *IEEE Transactions on Software Engineering*, SE-9, 6, (November 1983), 639–648. DOI: `10.1109/TSE.1983.235271`.

[6] Giuliano Antoniol, Aniello Cimitile, Giuseppe A. Di Lucca, and Massimiliano Di Penta. 2004. Assessing staffing needs for a software maintenance project through queuing simulation. *IEEE Transactions on Software Engineering*, 30, 1, (January 2004), 43–58. ISSN: 0098-5589. DOI: `10.1109/TSE.2004.1265735`.

[7] James Bach. 1995. Enough about process: what we need are heroes. *IEEE Software*, 12, 2, 96–98. ISSN: 0740-7459. DOI: `10.1109/52.368273`.

[8] Jerry Banks, John S. Carson II, Barry L. Nelson, and David M. Nicol. 2005. *Discrete-Event System Simulation.* (4th edition). Prentice Hall, Upper Saddle River, NJ.

[9] Kent Beck. 1999. Embracing change with extreme programming. *Computer*, 32, 10, 70–77. DOI: 10.1109/2.796139.

[10] Barry Boehm and Richard Turner. 2004. Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Driven Methods. In *Proceedings of the 26th International Conference on Software Engineering (ICSE), 2004.* IEEE, Edinburgh, UK, (May 2004), 718–719. DOI: 10.1109/ICSE.2004.1317503.

[11] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. 2007. *Programming Multi-agent Systems in AgentSpeak Using Jason.* English. (1 edition edition). Wiley-Blackwell, Chichester, (October 2007). ISBN: 978-0-470-02900-8.

[12] Anna Borjesson and Lars Mathiassen. 2003. Making SPI happen: the IDEAL distribution of effort. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the.* (January 2003). DOI: 10.1109/HICSS.2003.1174899.

[13] 2011. *Business Process Model and Notation (BPMN).* (2nd edition). Object Management Group, Inc., (January 2011). http://www.omg.org/spec/BPMN/2.0.

[14] Esther Cameron and Mike Green. 2009. *Making Sense of Change Management: A Complete Guide to the Models, Tools and Techniques of Organizational Change.* (2nd edition). Kogan Page Publishers, Philadelphia, PA, USA. ISBN: 978-0-7494-5310-7.

[15] Aaron G. Cass, Barbara Staudt Lerner, Stanley M. Sutton Jr, Eric K. McCall, Alexander Wise, and Leon J. Osterweil. 2000. Little-JIL/Juliette: a process definition language and interpreter. In *Proceedings of the 22nd international conference on Software engineering (ICSE), 2000.* ACM, Limerick, Ireland, 754–757. ISBN: 1-58113-206-9. DOI: 10.1145/337180.337623.

[16] 2013. Chaos Manifesto 2013: Think Big, Act Small. Technical report. The Standish Group International, Inc., 48. https://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf.

[17] Bin Chen. 2011. *Improving Processes Using Static Anlaysis Techniques.* PhD thesis. University of Massachusetts, Amherst, Amherst, MA, (February 2011). https://scholarworks.umass.edu/open_access_dissertations/332.

[18] Laku Chidambaram and Lai Lai Tung. 2005. Is Out of Sight, Out of Mind? An Empirical Study of Social Loafing in Technology-Supported Groups. *Information Systems Research*, 16, 2, (June 2005), 149–168. DOI: 10.1287/isre.1050.0051.

[19] Gerhard Chroust. 2000. Software Process Models: Structure and Challenges. In *Proceedings of the Conference on Software: Theory and Practice*. Yulin Feng, David Notkin, and Marie-Claude Gaudel, editors. Publishing House of Electronics Industry, Beijing, China, (August 2000), 279–286. ISBN: 7-5053-6110-4.

[20] Jamieson M. Cobleigh, Lori A. Clark, and Leon J. Osterweil. 2000. Verifying properties of process definitions. In *Proceedings of the 2000 ACM SIGSOFT international Symposium on Software Testing and Analysis (ISSTA)*. Volume 25. ACM, (September 2000), 96–101. DOI: 10.1145/347636.348876.

[21] Alistair Cockburn. 1999. Characterizing people as non-linear, first-order components in software development. Technical report HaT Technical Report 1999.03. Humans and Technology, (October 1999). Retrieved 04/10/2013 from http://alistair.cockburn.us/Characterizing+people+as+non-linear,+first-order+components+in+software+development/v/slim.

[22] Alistair Cockburn. 2004. Earned-value and Burn Charts. Technical Report HaT Technical Report TR 2004.04. (June 2004). Retrieved 06/22/2017 from http://alistair.cockburn.us/Earned-value+and+burn+charts.

[23] Alistair Cockburn. 2003. People and methodologies in software development. (February 2003). Retrieved 04/10/2013 from http://alistair.cockburn.us/People+and+methodologies+in+software+development.

[24] Alistair Cockburn. 2000. Selecting a project's methodology. *IEEE Software*, 17, 4, 64–71. DOI: 10.1109/52.854070.

[25] Mike Cohn. 2015. Multiple Levels of "Done" in Scrum. en. Blog. (February 2015). Retrieved 03/24/2019 from https://www.mountaingoatsoftware.com/blog/multiple-levels-of-done.

[26] Mike Cohn. 2012. Rotating The ScrumMaster Role. en. Blog. (January 2012). Retrieved 08/22/2018 from https://www.mountaingoatsoftware.com/blog/rotating-the-scrummaster-role.

[27] Richard M. Crowder, Mark A. Robinson, Helen P. N. Hughes, and Yee-Wai Sim. 2012. The Development of an Agent-Based Modeling Framework for Simulating Engineering Team Work. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 42, 6, (November 2012), 1425–1439. ISSN: 1083-4427. DOI: 10.1109/TSMCA.2012.2199304.

[28] Bill Curtis, Marc I. Kellner, and Jim Over. 1992. Process modeling. *Commun. ACM*, 35, 9, (September 1992), 75–90. ISSN: 0001-0782. DOI: 10.1145/130994.130998.

[29] Ian J. De Silva, Sanjai Rayadurgam, and Mats P. E. Heimdahl. 2015. A Reference Model for Simulating Agile Processes. In *Proceedings of the 2015 International Conference on Software and System Process* (ICSSP 2015). ACM, Tallinn, Estonia, (August 2015), 82–91. ISBN: 978-1-4503-3346-7. DOI: 10.1145/2785592.2785615.

[30] Ian J. De Silva, Sanjai Rayadurgam, and Mats P. E. Heimdahl. 2017. Domain Modeling for Development Process Simulation. In *Proceedings of the 2017 International Conference on Software and System Process* (ICSSP 2017). ACM, Paris, France, (July 2017). ISBN: 978-1-4503-5270-3. DOI: 10.1145/3084100.3084111.

[31] Tom DeMarco and Timothy R. Lister. 1999. *Peopleware: Productive Projects and Teams*. (2nd edition). Dorset House Pub. ISBN: 978-0-932633-43-9.

[32] Paolo Donzelli and Giuseppe Iazeolla. 2001. Hybrid simulation modelling of the software process. *Journal of Systems and Software*, 59, 3, (December 2001), 227–235. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(01)00064-4.

[33] Tore Dybå, Erik Arisholm, Dag I. K. Sjøberg, Jo E. Hannay, and Forrest Shull. 2007. Are Two Heads Better than One? On the Effectiveness of Pair Programming. *IEEE Software*, 24, 6, (November 2007), 12–15. ISSN: 0740-7459. DOI: 10.1109/MS.2007.158.

[34] Joshua M. Epstein. 2006. *Generative Social Science: Studies in Agent-Based Computational Modeling*. Princeton University Press, Princeton, NJ, USA. ISBN: 978-0-691-12547-3.

[35] Joshua M. Epstein and Robert L. Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up*. MIT press, Washington, DC, USA, (November 1996). ISBN: 0-262-05053-6.

[36] Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. 2009. Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. en. In *Enterprise, Business-Process and Information Systems Modeling*. Lecture Notes in Business Information Processing. Springer, Berlin, Heidelberg, (June 2009), 353–366. ISBN: 978-3-642-01862-6. DOI: 10.1007/978-3-642-01862-6_29.

[37] Jay Fields, Shane Harvie, Martin Fowler, and Kent Beck. 2009. *Refactoring: Ruby Edition*. English. (1 edition edition). Addison-Wesley Professional, Upper Saddle River, NJ, (October 2009). ISBN: 978-0-321-98413-5.

[38] Douglas B. Fridsma. 2003. *Organizational Simulation of Medical Work: An Information-Processing Approach*. PhD thesis. Standford University.

[39] Rosanna Garcia. 2005. Uses of Agent-Based Modeling in Innovation/New Product Development Research. *Journal of Product Innovation Management*, 22, 5, 380–398. ISSN: 1540-5885. DOI: 10.1111/j.1540-5885.2005.00136.x.

[40] Michael P. Georgeff and Amy L. Lansky. 1986. Procedural knowledge. *Proceedings of the IEEE*, 74, 10, 1383–1398. DOI: 10.1109/PROC.1986.13639.

[41] Michael P. Georgeff and Amy L. Lansky. 1987. Reactive reasoning and planning. In *AAAI*. Volume 87, 677–682.

[42] Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated planning: theory & practice*. Elsevier, San Francisco, CA, USA. ISBN: 1-55860-856-7.

[43] Michael E. Gorman. 2002. Types of knowledge and their roles in technology transfer. *The Journal of Technology Transfer*, 27, 3, 219–231. DOI: 10.1023/A:1015672119590.

[44] Anton Frank Harmsen. 1997. *Situational Method Engineering*. PhD thesis. University of Twente, Utrecht, NL. Retrieved 01/04/2017 from http://eprints.eemcs.utwente.nl/17266/.

[45] Brian Henderson-Sellers and Jolita Ralyté. 2010. Situational Method Engineering: State-of-the-Art Review. *Journal of Universal Computer Science*, 16, 3, 424–478. DOI: 10.3217/jucs-016-03-0424.

[46] Joseph Hurtado. 2013. Open Kanban, v1.00 Rev a. (August 2013). Retrieved 12/26/2018 from https://github.com/agilelion/Open-Kanban.

[47] 2006. IEEE Standard for Developing a Software Project Life Cycle Process. *IEEE Std 1074-2006 (Revision of IEEE Std 1074-1997)*, (July 2006), 1–110. DOI: 10.1109/IEEESTD.2006.219190.

[48] [n. d.] IFPUG - International Function Point Users Group. (). Retrieved 03/13/2015 from http://www.ifpug.org/.

[49] 2017. ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. International Standard ISO/IEC/IEEE 24765:2017(E). Geneva, Switzerland, (August 2017), 1–541. 10.1109/IEEESTD.2017.8016712.

[50] Ron Jefferies. 2004. A Metric Leading to Agility. Blog. (June 2004). Retrieved 12/26/2018 from https://ronjeffries.com/xprog/articles/jatrtsmetric/.

[51] Nicholas R. Jennings. 2000. On agent-based software engineering. *Artificial intelligence*, 117, 2, 277–296. DOI: 10.1016/S0004-3702(99)00107-1.

[52] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. 1998. A Roadmap of Agent Research and Development. en. *Autonomous Agents and Multi-Agent Systems*, 1, 1, (March 1998), 7–38. ISSN: 1573-7454. DOI: 10.1023/A:1010090405266.

[53] Yan Jin and Raymond E. Levitt. 1996. The virtual design team: a computational model of project organizations. en. *Comput Math Organiz Theor*, 2, 3, (September 1996), 171–195. ISSN: 1381-298X, 1572-9346. DOI: 10.1007/BF00127273.

[54] Georg Kalus and Marco Kuhrmann. 2013. Criteria for software process tailoring: a systematic review. In *Proceedings of the 2013 International Conference on Software and System Process*. ACM, 171–180. DOI: 10.1145/2486046.2486078.

[55] Stephen H. Kan. 2003. *Metrics and Models in Software Quality Engineering*. (2nd edition). Addison-Wesley, Boston, MA, USA. ISBN: 978-0-201-72915-3. Retrieved 02/09/2015 from.

[56] Marc I Kellner, Raymond J Madachy, and David M Raffo. 1999. Software process simulation modeling: Why? What? How? *Journal of Systems and Software*, 46, 2–3, (April 1999), 91–105. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(99)00003-5.

[57] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29, 6, 18–21. DOI: 10.1109/MS.2012.167.

[58] John C. Kunz, Tore R. Christiansen, Geoff P. Cohen, Yan Jin, and Raymond E. Levitt. 1998. The Virtual Design Team. *Communications of the ACM*, 41, 11, 84–91. DOI: 10.1145/287831.287844.

[59] Jo Ann Lane and Richard Turner. 2013. Improving Development Visibility and Flow in Large Operational Organizations. en. In *Lean Enterprise Software and Systems* (Lecture Notes in Business Information Processing). Springer, Berlin, Heidelberg, (December 2013), 65–80. ISBN: 978-3-642-44930-7. DOI: 10.1007/978-3-642-44930-7_5.

[60] Jintae Lee and George M. Wyner. 2003. Defining specialization for dataflow diagrams. *Information Systems*, 28, 6, (September 2003), 651–671. ISSN: 0306-4379. DOI: 10.1016/S0306-4379(02)00044-3.

[61] Meir M. Lehman, Goel Kahen, and Juan F. Ramil. 2005. Simulation Process Modelling for Managing Software Evolution. In *Software Process Modeling*. Number 10 in International Series in Software Engineering. Dr Silvia T. Acuña and Dr Natalia Juristo, editors. Springer US, (January 2005), 87–109. ISBN: 978-0-387-24262-0. DOI: 10.1007/0-387-24262-7_4.

[62] Raymond E. Levitt. 2012. The Virtual Design Team: Designing Project Organizations as Engineers Design Bridges. *Journal of Organization Design*, 1, 2, (August 2012), 14. ISSN: 2245-408X. DOI: 10.7146/jod.6345.

[63] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101, (March 2015), 193–220. DOI: 10.1016/j.jss.2014.12.027.

[64] Ray Madachy and Denton Tarbet. 1999. Initial Experiences in Software Process Modeling. (1999). https://csse.usc.edu/csse/event/1999/COCOMO/10_Madachy%20Initial.pdf.

[65] Raymond J. Madachy. 1996. System Dynamics Modeling of an Inspection-Based Process. In *Proceedings of the 18th International Conference on Software Engineering (ICSE), 1996*. IEEE, Berlin, DE, (March 1996), 376–386. DOI: 10.1109/ICSE.1996.493432.

[66] 2001. Manifesto for Agile Software Development. (2001). Retrieved 10/06/2014 from http://agilemanifesto.org/.

[67] Robert H. Martin and David Raffo. 2000. A Model of the Software Development Process Using Both Continuous and Discrete Models. en. *Software Process: Improvement and Practice*, 5, 2-3, 147–157. ISSN: 1099-1670. DOI: 10.1002/1099-1670(200006/09)5:2/3<147::AID-SPIP122>3.0.CO;2-T.

[68] Robert Martin and David Raffo. 2001. Application of a hybrid process simulation model to a software development project. *Journal of Systems and Software*, 59, 3, 237–246. DOI: 10.1016/S0164-1212(01)00065-6.

[69] Juan Martínez-Miranda, Arantza Aldea, René Bañares-Alcántara, and Matías Alvarado. 2006. TEAKS: simulation of human performance at work to support team configuration. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems* (AAMAS '06). ACM, New York, NY, USA, 114–116. ISBN: 1-59593-303-4. DOI: 10.1145/1160633.1160649.

[70] Juan Martínez-Miranda and Juan Pavón. 2012. Modeling the influence of trust on work team performance. en. *SIMULATION*, 88, 4, (April 2012), 408–436. ISSN: 0037-5497, 1741-3133. DOI: 10.1177/0037549711404714.

[71] Juan Martínez-Miranda and Juan Pavón. 2009. Modelling Trust into an Agent-Based Simulation Tool to Support the Formation and Configuration of Work Teams. en. In *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009)*. Number 55 in Advances in Intelligent and Soft Computing. Yves Demazeau, Juan Pavón, Juan M. Corchado, and Javier Bajo, editors. Springer Berlin Heidelberg, 80–89. ISBN: 978-3-642-00487-2. DOI: 10.1007/978-3-642-00487-2_9.

[72] Steve McConnell. 2008. Right-Sizing Agile Development. Webinar. (October 2008). http://www.computer.org/portal/web/newwebinars/mcconnell.

[73] Peiwei Mi and Walt Scacchi. 1990. A knowledge-based environment for modeling and simulating software engineering processes. *IEEE Transactions on Knowledge and Data Engineering*, 2, 3, (September 1990), 283–289. ISSN: 1041-4347. DOI: 10.1109/69.60792.

[74] Peiwei Mi and Walt Scacchi. 1993. Articulation: an integrated approach to the diagnosis, replanning, and rescheduling of software process failures. In *Proceedings of 8th Knowledge-Based Software Engineering Conference*. (September 1993), 77–84. DOI: 10.1109/KBSE.1993.341195.

[75] Dominic Müller, Manfred Reichert, and Joachim Herbst. 2006. Flexibility of Data-Driven Process Structures. en. In *Business Process Management Workshops* (Lecture Notes in Computer Science). Johann Eder and Schahram Dustdar, editors. Springer Berlin Heidelberg, 181–192. ISBN: 978-3-540-38445-8.

[76] Mark Müller and Dietmar Pfahl. 2008. Simulation Methods. In *Guide to Advanced Empirical Software Engineering*. Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors. Springer London, 117–152. ISBN: 978-1-84800-044-5. DOI: 10.1007/978-1-84800-044-5_5.

[77] Jürgen Münch, Ove Armbrust, Martin Kowalczyk, Martín Soto, Jürgen Münch, Ove Armbrust, Martin Kowalczyk, and Martín Soto. 2012. Software Process Simulation. In *Software Process Definition and Management*. The Fraunhofer IESE Series on Software and Systems Engineering. Dieter Rombach and Peter Liggesmeyer, editors. Springer Berlin Heidelberg, 187–210. ISBN: 978-3-642-24291-5. DOI: 10.1007/978-3-642-24291-5_7.

[78] Jürgen Münch, Dietmar Pfahl, and Ioana Rus. 2005. Virtual software engineering laboratories in support of trade-off analyses. *Software Quality Journal*, 13, 4, 407–428. 00010. DOI: 10.1007/s11219-005-4253-y.

[79] Jeff Patton. 2014. Read This First. In *User Story Mapping: Discover the Whole Story, Build the Right Product*. (1st edition). O'Reilly Media, Inc., Sebastopol, CA, (September 2014). ISBN: 978-1-4919-0490-9. Retrieved 12/26/2018 from http://www.jpattonassociates.com/read-this-first/.

[80] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. 2007. DECLARE: Full Support for Loosely-Structured Processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. (October 2007), 287–287. DOI: 10.1109/EDOC.2007.14.

[81] M. Pesic and W. M. P. van der Aalst. 2006. A Declarative Approach for Flexible Business Processes Management. en. In *Business Process Management Workshops* (Lecture Notes in Computer Science). Johann Eder and Schahram Dustdar, editors. Springer Berlin Heidelberg, 169–180. ISBN: 978-3-540-38445-8.

[82] Mary Poppendieck and Tom Poppendieck. 2003. *Lean Software Development: An Agile Toolkit*. Addison-Wesley.

[83] 2011. *Practice Standard for Earned Value Management*. Project Management Institute, Incorporated. https://app.knovel.com/hotlink/toc/id:kpPSEVME01/practice-standard-earned/practice-standard-earned.

[84] Lawrence H. Putnam and Ware Myers. 2003. *Five Core Metrics*. Dorset House Publishing, New York, NY, USA. ISBN: 978-0-932633-55-2.

[85] David Raffo, Timo Kaltio, Derek Partridge, Keith Phalp, and Juan F. Ramil. 1999. Empirical Studies Applied to Software Process Models. *Empirical Software Engineering*, 4, 4, 353–369. ISSN: 1382-3256. DOI: 10.1023/A:1009817721252.

[86] Jolita Ralyté, Rébecca Deneckère, and Colette Rolland. 2003. Towards a Generic Model for Situational Method Engineering. en. In *Advanced Information Systems Engineering*. Springer, Berlin, Heidelberg, (June 2003), 95–110. DOI: 10.1007/3-540-45017-3_9.

[87] Jolita Ralyté and Colette Rolland. 2001. An Approach for Method Reengineering. en. In *SpringerLink* (Lecture Notes in Computer Science). Springer Berlin Heidelberg, (November 2001), 471–484. DOI: 10.1007/3-540-45581-7_35.

[88] M. Raunak, L. Osterweil, A. Wise, L. Clarke, and P. Henneman. 2009. Simulating patient flow through an Emergency Department using process-driven discrete event simulation. In *ICSE Workshop on Software Engineering in Health Care, 2009. SEHC '09*. (May 2009), 73–83. DOI: 10.1109/SEHC.2009.5069608.

[89] Pierre N. Robillard. 1999. The Role of Knowledge in Software Development. *Commun. ACM*, 42, 1, (January 1999), 87–92. ISSN: 0001-0782. DOI: 10.1145/291469.291476.

[90] Colette Rolland and Naveen Prakash. 1996. A Proposal For Context-Specific Method Engineering. en. In *Method Engineering* (IFIP — The International Federation for Information Processing). Springer, Boston, MA, (August 1996), 191–208. ISBN: 978-0-387-35080-6. DOI: 10.1007/978-0-387-35080-6_13.

[91] Winston W. Royce. 1987. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering* (ICSE '87). IEEE Computer Society Press, Los Alamitos, CA, USA, 328–338. ISBN: 0-89791-216-0. Retrieved 03/29/2013 from http://dl.acm.org/citation.cfm?id=41765.41801.

[92] Kenneth S. Rubin. 2012. *Essential Scrum: A practical guide to the most popular Agile process*. Addison-Wesley, Ann Arbor, MI, USA. ISBN: 978-0-13-704329-3.

[93] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, Hoboken, NJ, USA. ISBN: 978-1-118-10435-4.

[94] Ioana Rus, James Collofello, and Peter Lakey. 1999. Software process simulation for reliability management. *Journal of Systems and Software*, 46, 2, (April 1999), 173–182. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(99)00010-2.

[95] Nick Russell, Wil van der Aalst, and Arthur ter Hofstede. 2006. Workflow Exception Patterns. en. In *Advanced Information Systems Engineering*. Springer, Berlin, Heidelberg, (June 2006), 288–302. DOI: 10.1007/11767138_20.

[96] Stuart Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach*. (3rd edition). Prentice Hall, Upper Saddle River, NJ. ISBN: 978-0-13-604259-4.

[97] Walt Scacchi. 1999. Experience with software process simulation and modeling. *Journal of Systems and Software*, 46, 2–3, (April 1999), 183–192. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(99)00011-4.

[98] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil Aalst. 2008. Process Flexibility: a Survey of Contemporary Approaches. en. In *Advances in Enterprise Engineering I*. Number 10 in Lecture Notes in Business Information Processing. Jan L. G. Dietz, Antonia Albani, and Joseph Barjis, editors. Springer Berlin Heidelberg, 16–30. ISBN: 978-3-540-68643-9. DOI: 10.1007/978-3-540-68644-6_2.

[99] Ken Schwaber and Jeff Sutherland. 2017. The Scrum Guide: The Definitive Guide to Scrum. (2017). https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf.

[100] James Shore and Shane Warden. 2008. *The Art of Agile Development*. English. (1 edition edition). O'Reilly Media, Sebastopol, CA. ISBN: 978-0-596-52767-9.

[101] P. O. Siebers, C. M. Macal, J. Garnett, D. Buxton, and M. Pidd. 2010. Discrete-event simulation is dead, long live agent-based simulation! en. *Journal of Simulation*, 4, 3, (September 2010), 204–210. ISSN: 1747-7778, 1747-7786. DOI: 10.1057/jos.2010.14.

[102] Yee Wai Sim, Richard Crowder, Mark Robinson, and Helen Hughes. 2009. An Agent-based approach to modelling integrated product teams undertaking a design activity. In *Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*. Volume 2. ASME, San Diego, CA, (September 2009), 227–236. ISBN: 978-0-7918-4899-9. DOI: 10.1115/DETC2009-86957.

[103] 2008. Software & Systems Process Engineering Meta-Model Specification - Version 2.0. Standard formal/2008-04-01. Object Management Group, (April 2008). Retrieved 12/14/2017 from https://www.omg.org/spec/SPEM/2.0.

[104] Andreas Spillner, Tilo Linz, and Hans Schaefer. 2014. Fundamentals of Testing. In *Software Testing Foundations*. (4th edition). Rocky Nook Inc., Santa Barbra, CA. ISBN: 978-1-937538-42-2.

[105] Jonette M. Stecklein, Jim Dabney, Brandon Dick, Bill Haskins, Randy Lovell, and Gregory Moroney. 2004. Error Cost Escalation through the Project Life Cycle. Conference Paper JSC-CN-8435. NASA, Houston, TX, (June 2004). Retrieved 10/05/2015 from http://ntrs.nasa.gov/search.jsp?R=20100036670.

[106] Andrew S. Tanenbaum and Maarten Van Steen. 2007. *Distributed systems: principles and paradigms*. Prentice-Hall.

[107] Jan Thomsen, Raymond E. Levitt, and Clifford I. Nass. 2005. The Virtual Team Alliance (VTA): Extending Galbraith's Information-Processing Model to Account for Goal Incongruency. *Comput. Math. Organ. Theory*, 10, 4, (January 2005), 349–372. ISSN: 1381-298X. DOI: 10.1007/s10588-005-6286-y.

[108] Alexey Tregubov and Jo Ann Lane. 2015. Simulation of Kanban-based Scheduling for Systems of Systems: Initial Results. *Procedia Computer Science*. 2015 Conference on Systems Engineering Research 44, (January 2015), 224–233. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.03.004.

[109] Richard Turner, Levent Yilmaz, Jeffrey Smith, Donghuang Li, Saicharan Chada, Alice Smith, and Alexey Tregubov. 2016. DATASEM: a Simulation Suite for SoSE Management Research. In *2016 11th System of Systems Engineering Conference (SoSE)*. (June 2016), 1–6. DOI: 10.1109/SYSOSE.2016.7542954.

[110] Richard Turner, Levent Yilmaz, Jeffrey Smith, Donghuang Li, Saicharan Chada, Alice Smith, and Alexey Tregubov. 2015. Modeling an Organizational View of the SoS Towards Managing its Evolution. In *2015 10th System of Systems Engineering Conference (SoSE)*. (May 2015), 480–485. DOI: 10.1109/SYSOSE.2015.7151915.

[111] John D. Tvedt and James S. Collofello. 1995. Evaluating the Effectiveness of Process Improvements on Software Development Cycle Time via System Dynamics Modeling. *Proceedings of the International Computer Software and Applications Conference (COMPSAC) 19*, 318–325. http://130.203.133.150/viewdoc/summary?doi=10.1.1.53.8009.

[112] P Wernick and M.M Lehman. 1999. Software process white box modelling for FEAST/1. *Journal of Systems and Software*, 46, 2-3, 193–201. ISSN: 01641212. DOI: 10.1016/S0164-1212(99)00012-6.

[113] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. 2000. Strengthening the Case for Pair Programming. *IEEE Software*, 17, 4, (July 2000), 19–25. ISSN: 0740-7459. DOI: 10.1109/52.854064.

[114] Alexander Wise. 2006. Little-JIL 1.5 Language Report. Technical report UM-CS-2006-51. University of Massachusetts, Amherst, Amherst, MA, (October 2006), 1–28. http://laser.cs.umass.edu/techreports/06-51.pdf.

[115] Alexander Wise, Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Stanley M. Sutton Jr. 2000. Using Little-JIL to coordinate agents in software engineering. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. IEEE, Grenoble, France, 155–163. ISBN: 978-0-7695-0710-1. DOI: 10.1109/ASE.2000.873660.

[116] Michael Wooldridge. 2013. Intelligent Agents. In *Multiagent Systems*. (2nd edition). Gerhard Weiss, editor. The MIT Press, Cambridge, Massachusetts, (March 2013), 1–50. ISBN: 978-0-262-01889-0.

[117] Michael Wooldridge and Nicholas R. Jennings. 1995. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10, 02, 115–152. DOI: 10.1017/S0269888900008122.

[118] Edward Yourdon. 2004. *Death March*. (2nd edition). Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, USA. ISBN: 0-13-143635-X.