Domain Modeling for Development Process Simulation

Ian J. De Silva
Dept. of Comp. Sci. and Engr.
University of Minnesota, USA
desilva@cs.umn.edu

Sanjai Rayadurgam
Dept. of Comp. Sci. and Engr.
University of Minnesota, USA
rsanjai@cs.umn.edu

Mats P. E. Heimdahl
Dept. of Comp. Sci. and Engr.
University of Minnesota, USA
heimdahl@cs.umn.edu

ABSTRACT

Simulating agile processes prior to adoption can reduce the risk of enacting an ill-fitting process. Agent-based simulation is well-suited to capture the individual decision-making valued in agile. Yet, agile's lightweight nature creates simulation difficulties as agents must fill-in gaps within the specified process. Deliberative agents can do this given a suitable planning domain model. However, no such model, nor guidance for creating one, currently exists.

In this work, we propose a means for constructing an agile planning domain model suitable for agent-based simulation. As such, the domain model must ensure that all activity sequences derived from the model are executable by a software agent. We prescribe iterative elaboration and decomposition of an existing process to generate successive internally-complete and -consistent domain models, thereby ensuring plans derived from the model are valid. We then demonstrate how to generate a domain model and exemplify its use in planning the actions of a single agent.

CCS CONCEPTS

•**Computing methodologies** → **Modeling methodologies;** *Agent / discrete models;* Planning and scheduling; •**Software and its engineering** → *Agile software development;*

KEYWORDS

Domain Modeling, Software Development Process Evaluation

ACM Reference format:
Ian J. De Silva, Sanjai Rayadurgam, and Mats P. E. Heimdahl. 2017. Domain Modeling for Development Process Simulation. In *Proceedings of 2017 International Conference on Software and Systems Process, Paris, France, July 2017 (ICSSP'17),* 5 pages.
DOI: 10.1145/3084100.3084111

1 INTRODUCTION

Software development processes do not ensure project success, yet adopting an ill-fitting process may hinder the team's ability to complete the project on-time, on-budget, and with the required functionality. Prior to adoption, we want to evaluate candidate processes, particularly agile processes, to ensure they satisfy the project goals. Agent-based simulation is well-suited to such evaluation [3].

Because of their people-focused, lightweight nature, agile processes present unique difficulties for simulation. Rather than specifying all actions, lightweight processes specify some actions or quality criteria and allow individuals to supplement the process (choosing additional steps that both satisfy the criteria and reach the goal). Existing simulations cannot replicate this behavior, and, instead, rely on full process specification [3]. We desire a simulation that can model lightweight processes and capture the impact of individual decisions on a project and its outcomes.

Deliberative software agents[1] can model process supplementation by composing plans (sequences of process activities) expected to reach a goal state, then selecting and executing the most desirable (and process compliant) plan. Execution stops when the agent reaches the goal state or the plan becomes impracticable—wherein it forms a new plan. A planner module within the agent forms these plans over a domain model (the set of activities the agent can perform and their sequencing constraints). Unfortunately, there are no existing planning domain models nor guidance for creating them in the literature. The *objective of this paper*, is to provide guidance for creating a planning domain 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 -consistent (for each plan activity, all activities it depends on precede it in the plan).[2] Thus, with a cooperative planner, invalid plans are only generated when the domain model contains internal incompleteness or inconsistency. We, therefore, desire a 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[3] 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 demonstrate this approach by constructing an example planning domain model and use it to generate goal-reaching plans without a specified process.

ICSSP'17, Paris, France

978-1-4503-5270-3/17/07…$15.00
DOI: 10.1145/3084100.3084111

[1] A deliberative agent has an "explicitly represented, symbolic model of the world, and in which decisions (for example, about which actions to perform) are made via [logical] reasoning, based on pattern matching and symbolic manipulation" [14].
[2] There are other situation-independent process quality criteria [6, 7]. However, they address process desirability rather than validity, and are unrelated to plan formation.
[3] We assume the agent has perfect knowledge (full world observability).

2 RELATED WORK

Constructing Processes. Situational method engineering is a means for constructing situation-specific, fully-specified processes (methods) from process fragments stored in a repository. A process may be generated using method assembly, method configuration, or paradigm-based approaches [7]. Method assembly composes fragments from a repository according to the requirements/goals they satisfy [10]. Method configuration, including process tailoring, transforms an existing process specification by adding, removing, or elaborating it, according to guides/patterns, using fragments from a repository [7]. Paradigm-based construction generates/uses a meta-model that it transforms to meet the situational requirements then instantiates with fragments from a repository [10]. While all of these approaches generate situation-specific processes, they rely on the repository data's quality. If needed fragments are missing, it may not be possible to construct a valid process.

Situational method engineering aims to generate good processes for human actors to follow. While useful as guidance, this is insufficient for approximating human behaviors as humans are not well-behaved [9]. Humans, due to external factors, perform redundant activities or compose activities in unusual ways. We want to capture realistic behavior as we expect it will improve a simulation's predictive qualities. Rather than describing all acceptable ways in which an agent may deviate from the ideal, which may miss realistic behaviors, we want to permit all valid behaviors and allow the agent to reason over them.

Structured Process Transformation. Supporting process construction, Chroust introduces a model calculus to express process transformations [2], yet he does not show that these operations result in valid process models. Lee and Wyner define formal semantics for extending (specializing and refining) existing processes captured as data flow diagrams [8]. However, their approach is flow-preserving. Rather than adhering to fixed, idealized flows, we want agents to use any valid, process-constraint-complying activity sequencing to achieve the goal.

3 PLANNING DOMAIN MODEL CREATION

We describe an approach for generating a planning domain model and show that following this method ensures the result is ICC. We also provide guidance for specifying achievable intermediate goals despite possible activity output non-determinism within the model.

3.1 Constructing the Domain Model

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., [1, 12])—contain a wealth of method information that can aid our elaboration. Using these as fragment sources, we have summarized our method in Listing 1.

Listing 1: Our approach expressed algorithmically.

```
while(not shouldStop(domainModel)):
  (activity, fragment) = locateReplacement(domainModel)
  fragment = generalize(fragment)
  domainModel = replace(domainModel, activity, fragment)
```

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.[4]

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.

3.2 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 specifications' and domain model's ICC properties.

3.2.1 Definitions. A *process (or 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.

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

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

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}.

3.2.2 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, 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 $\mathcal{P}(V_P)$ where any activity's 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.

3.2.3 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'.

3.3 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.

3.3.1 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.

3.3.2 Intermediate Goal Planning. By replannability, we can reach the domain model goal state from any reachable world, However, not all teams want to reach that goal. We'd like to define other, intermediate goals and plan to them.

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 generation, we can express IGs only over those data objects we could arrive at deterministically. This severely limits our model. 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 execution (e.g., a defect in compiled code), or imperfect world knowledge (e.g., an artifact that is unexpectedly missing).[5] By our earlier assumptions—that we are simulating a single agent with perfect knowledge—the latter cannot be true, thus exceptional output must stem 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 1), the expected output depends on the world's state (e.g, tests 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 [5]. 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 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]Several forms of workflow faults exists [11]; however, in terms of expected activity output, only these apply.

4 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.

4.1 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 1).

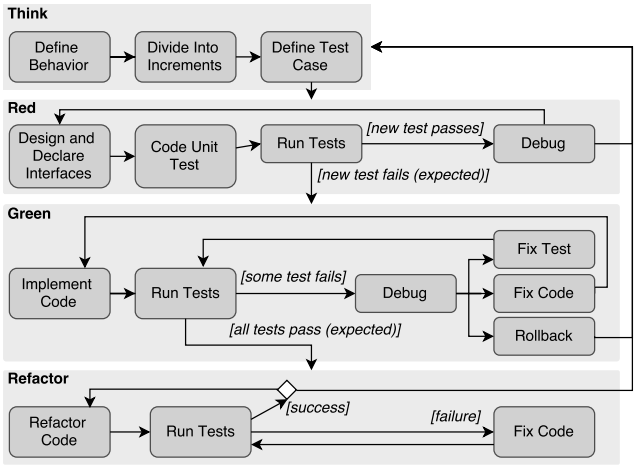

Figure 1: TDD – Detailed Control Flow [12]

Initial Domain Model. In order to simulate, we require a domain model. Here, we outline the steps needed to create one suitable for representing TDD.

Begin with an initial domain model: `software development`. Of the numerous possible elaborations, we select the V-model [13] since TDD is focused on testing. We expand `implementation` using IEEE Standard 1074 [1] and introduce activities for declaring, implementing, and integrating interfaces as prescribed by TDD for testing [12]. Spillner et al. [13] provides a general testing process that elaborates each of the testing activities. To support TDD's test automation, we replace `test coding` with the implementation process from earlier and `test execution` with TDD's definition [12]. Finally, we introduce refactoring according to Fields et al.'s description [4]. Listing 2 shows a subset of our domain model.

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 1).

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).

Listing 2: Domain Model Subset – Activities are defined as `<activityName>: <preconditions> -> <postconditions>`, where capitalized terms are variables to unify. Activities starting with `trans` split data objects to constituent parts.

```
technicalSysDesign: artifact(requirement) -> artifact(sysArch)
trans_ArchToComp: artifact(sysArch) -> artifact(componentDefinition),
      artifact(componentIntegrationModel)
testPlanning: artifact(requirement) -> artifact(testDesign)
specifyComponent: artifact(componentDefinition) -> artifact(
      componentDesign), artifact(incrementRequirement)
declareInterface: artifact(incrementRequirement), artifact(
      componentDesign) -> artifact(incrementInterface)
implIncrement: artifact(incrementRequirement), artifact(
      incrementInterface) -> artifact(incrementImpl)
integrateIncrements: artifact(incrementImpl), artifact(
      componentDesign) -> artifact(componentImpl)
integrateComponents: artifact(componentImpl), artifact(
      componentDesign) -> artifact(sysImpl)
compileSys: artifact(Impl), isA(Impl, impl), artifact(compilationSys)
      -> artifact(compiledSys)
specifyUnitTests: artifact(Type), isA(Type, testBasis), artifact(
      incrementInterface), artifact(testDesign) -> artifact(
      logicalUnitTest)
genConcreteUnitTests: artifact(logicalUnitTest) -> artifact(
      concreteUnitTest)
implUnitTest: artifact(incrementInterface), artifact(
      concreteUnitTest), artifact(testingTool) -> artifact(
      automatedUnitTestScript)
integrateTest: artifact(automatedUnitTestScript), artifact(
      testDesign) -> artifact(testSuiteImpl)
compileTests: artifact(testSuiteImpl), artifact(testCompilationSys)
      -> artifact(compiledTestSuite)
executeTests: artifact(compiledTestSuite), artifact(compiledSys)->
      artifact(testExecutionResult)

isA(incrementRequirement, testBasis)
isA(sysImpl, impl)
isA(incrementInterface, impl)
```

Table 1: Mapping from TDD to the Domain Model Subset

TDD Activity	Domain Model Activity
Define Behavior, Divide into Increments	`specifyComponent`
Define Test Case	`specifyUnitTests`, `genConcreteUnitTests`
Design & Declare Interfaces	`declareInterface`
Code Unit Test	`implUnitTest`
Run Tests	`executeTests`
Implement Code	`implIncrement`

4.2 Simulating

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

4.2.1 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 future work.

Utility. Utility functions help the agent determine plan desirability. To capture process adherence, we biased behavior towards quickly completing orderings (pairs that represent activity partial orderings; i.e., succession) with the following utility function:

$$U(a) = \frac{2}{3}W(a) + \frac{1}{3}max(\ 0, U(successor(a))\ )$$

where a is the current activity in the plan, $successor(a)$ is the current activity's immediate successor in the plan, and $W(a)$ is the activity's weight based on initiating or completing an ordering (arriving at a node on the left or right side of a pair, respectively); defined as:

$$W(a) = \begin{cases} 1, & \text{if } a \text{ completes an ordering} \\ 0.5 & \text{if } a \text{ initiates an ordering but does not complete one} \\ 0 & \text{if } a \text{ does not initiate or complete an ordering} \end{cases}$$

To evaluate that an agent can reach the goal without full process specification, we defined a constant utility function and ran the agent with no specified process. Under these conditions, the agent should choose plans at random until it reaches the goal.

4.2.2 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 3) 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.

Listing 3: Simulations by Utility Function

(a) Constant-value	(b) Weighted: TDD	(c) Weighted: TLD
defineRequirements	defineRequirements	defineRequirements
technicalSysDesign	testPlanning	technicalSysDesign
testPlanning	technicalSysDesign	trans_ArchToComp
trans_ArchToComp	trans_ArchToComp	specifyComponent
specifyComponent	specifyComponent	declareInterface
declareInterface	declareInterface	implIncrement
compileSys	compileSys	integrateIncrements
implIncrement	specifyUnitTests	integrateComponents
specifyUnitTests	genConcreteUnitTests	compileSys
integrateIncrements	implUnitTest	testPlanning
integrateComponents	integrateTest	specifyUnitTests
genConcreteUnitTests	compileTests	genConcreteUnitTests
implUnitTest	executeTests	implUnitTest
integrateTest	implIncrement	integrateTest
compileTests	integrateIncrements	compileTests
compileSys	integrateComponents	executeTests
executeTests	compileSys	
	executeTests	

Threats to Validity. 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). As this is part of ongoing work, we are laboring towards a larger example that incorporates these concerns and extends to multi-agent simulations.

5 CONCLUSION AND FUTURE WORK

In this work, we presented a means for constructing a planning domain model; an important step towards simulating agile processes for a priori evaluation. We showed that iteratively elaborating a domain model preserves its internal-completeness and -consistency, ensuring plan validity when plans are generated by a cooperative planner. We then used the prescribed approach to construct an example domain model able to express test-driven development, and showed an agent can use the model to both adhere to a fully-specified process and achieve a goal absent a specified process.

Our approach generates domain models for an agent with perfect knowledge. As part of our on-going work, we expect to enhance the domain model to support multi-agent planning, paying particular attention to difficulties introduced by scaling (both in number of agents and domain model size) and partial world observability (imperfect knowledge). To this end, we are exploring a means to scale using intermediate goals to support hierarchical planning.

Our domain modeling approach lays the foundation for agent deliberation of software processes; providing a means for both generating plans without full process specification and for simulating agile processes.

REFERENCES

[1] 2006. IEEE Standard for Developing a Software Project Life Cycle Process. *IEEE Std 1074-2006* (July 2006), 1–110. DOI: http://dx.doi.org/10.1109/IEEESTD.2006.219190

[2] 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 (Eds.). PHEI, Beijing, China, 279 – 286.

[3] 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, 82–91. DOI: http://dx.doi.org/10.1145/2785592.2785615

[4] Jay Fields, Shane Harvie, Martin Fowler, and Kent Beck. 2009. *Refactoring: Ruby Edition* (1 ed.). Addison-Wesley Professional, Upper Saddle River, NJ.

[5] Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated planning: theory & practice*. Elsevier.

[6] Anton Frank Harmsen. 1997. Situational method engineering. (1997). http://eprints.eemcs.utwente.nl/17266/

[7] Brian Henderson-Sellers and Jolita Ralyté. 2010. Situational Method Engineering: State-of-the-Art Review. *J. UCS* 16, 3 (2010), 424–478. DOI: http://dx.doi.org/10.3217/jucs-016-03-0424

[8] Jintae Lee and George M. Wyner. 2003. Defining specialization for dataflow diagrams. *Information Systems* 28, 6 (Sept. 2003), 651–671. DOI: http://dx.doi.org/10.1016/S0306-4379(02)00044-3

[9] Nesi Outmazgin and Pnina Soffer. 2016. A process mining-based analysis of business process work-arounds. *Softw Syst Model* 15, 2 (May 2016), 309–323. DOI: http://dx.doi.org/10.1007/s10270-014-0420-6

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

[11] Nick Russell, Wil van der Aalst, and Arthur ter Hofstede. 2006. Workflow Exception Patterns. In *Advanced Information Systems Engineering*. Springer, Berlin, Heidelberg, 288–302. DOI: http://dx.doi.org/10.1007/11767138_20

[12] James Shore and Shane Warden. 2008. *The Art of Agile Development* (1 ed.). O'Reilly Media, Sebastopol, CA.

[13] Andreas Spillner, Tilo Linz, and Hans Schaefer. 2014. Fundamentals of Testing. In *Software Testing Foundations* (4 ed.). Rocky Nook Inc., Santa Barbra, CA.

[14] Michael Wooldridge and Nicholas R. Jennings. 1995. Intelligent agents: Theory and practice. *The knowledge engineering review* 10, 02 (1995), 115–152. DOI: http://dx.doi.org/10.1017/S0269888900008122