Bridging the Gap between Requirements Modeling and Behavior-driven Development

Mauricio Alferez*, Fabrizio Pastore*, Mehrdad Sabetzadeh*, Lionel C. Briand*†, Jean-Richard Riccardi§
*SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
†School of Engineering and Computer Science, University of Ottawa, Canada
§Clearstream Services SA, Luxembourg
Email: {alferez, pastore, sabetzadeh, briand}@svv.lu, jean-richard.riccardi@clearstream.com

***Abstract*—Acceptance criteria (AC) are implementation agnostic conditions that a system must meet to be consistent with its requirements and be accepted by its stakeholders. Each acceptance criterion is typically expressed as a natural-language statement with a clear pass or fail outcome. Writing AC is a tedious and error-prone activity, especially when the requirements specifications evolve and there are different analysts and testing teams involved. Analysts and testers must iterate multiple times to ensure that AC are understandable and feasible, and accurately address the most important requirements and workflows of the system being developed.**

In many cases, analysts express requirements through models, along with natural language, typically in some variant of the UML. AC must then be derived by developers and testers from such models. In this paper, we bridge the gap between requirements models and AC by providing a UML-based modeling methodology and an automated solution to generate AC. We target AC in the form of Behavioral Specifications in the context of Behavioral-Driven Development (BDD), a widely used agile practice in many application domains. More specially we target the well-known Gherkin language to express AC, which then can be used to generate executable test cases.

We evaluate our modeling methodology and AC generation solution through an industrial case study in the financial domain. Our results suggest that (1) our methodology is feasible to apply in practice, and (2) the additional modeling effort required by our methodology is outweighed by the benefits the methodology brings in terms of automated and systematic AC generation and improved model precision.

***Index Terms*—Software testing, BDD, modeling, requirements engineering, text generation, Gherkin, and FinTech.**

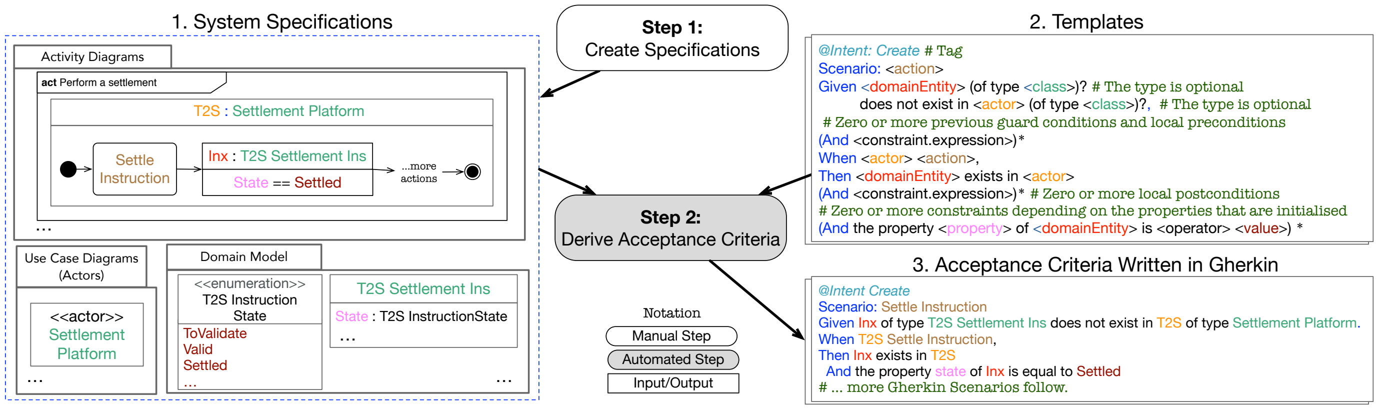
I. Introduction

Business-critical information systems, for example those used in banking and securities services, are often subject to extensive testing at different stages of development. *Acceptance testing* aims at ensuring that a completed system as a whole meets its specified requirements [1]. This type of testing typically involves users and other individuals who have strong domain knowledge. A key step in acceptance testing is to define *Acceptance Criteria (AC)* to distinguish correct behaviors from potentially incorrect ones. AC are typically written in natural language to facilitate communication among the stakeholders, from clients, to project management and to the development team. In the context of *Behavior-Driven Development* (BDD), the Gherkin language [2] is commonly used to express AC using a *Given-When-Then* structure: *Given* [initial context] *When* [event or action] *Then* [expected result]. For example, "*Given* there is enough money on my account *When* I make a withdrawal *Then* I get the expected amount of money from the ATM". This Given-When-Then structure is known as a *Gherkin scenario*. An acceptance criterion specification describes an execution scenario (hereafter, *acceptance testing scenario*), and is usually captured by means of a sequence of Gherkin scenarios. The popularity of the Gherkin language is, in large part, due to its ability to enforce the use of high-level, domain-specific terms and supporting traceability from AC to executable test cases [2]. More precisely, the Gherkin syntax enables the automated generation of executable test cases based on matching AC text to APIs, thus leading to test case traceability.

Information systems typically support and manage human processes, including multiple automated and manual tasks (e.g., evaluation of loan applications), and their requirements are often (partly) captured by means of behavioral models (e.g., UML activity diagrams). There is a disconnect between such requirements models, produced by analysts, and AC specifications, written by developers and testers in the context of BDD. Such specifications are manually derived from models in a tedious and error-prone process, in which there are many opportunities for misunderstanding. For example, activity diagrams are often partially specified and lack pre- and postconditions, thus complicating the identification of context information and expected results in AC. Also, being high-level, requirements models cannot help determine which are the outputs (e.g., GUI, logs) to be inspected during acceptance testing, thus preventing the clear specification of expected results in AC and complicating the implementation of executable test cases (e.g., to automatically load the generated results and print the test outcome). Finally, requirements models capture complex behaviors including blocking conditions and parallel flows – elements that may lead to incomplete identification of AC.

To address the above challenges, we propose an approach, named *AGAC (Automated Generation of Acceptance Criteria)*, which supports the automated generation of AC specifications in Gherkin. AGAC provides the following key benefits. First, requirements analysts no longer need to verify whether existing AC are still valid, since they are automatically generated. Second, AC are now produced systematically, thus decreasing the likelihood that critical system-level scenarios would be

Accepted for publication in the Proceedings of the 22nd IEEE / ACM International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE Computer Society, 2019



```
@Intent: Create # Tag
Scenario: <action>
Given <domainEntity> (of type <class>)? # The type is optional
        does not exist in <actor> (of type <class>)?,  # The type is optional
 # Zero or more previous guard conditions and local preconditions
(And <constraint.expression>)*
When <actor> <action>,
Then <domainEntity> exists in <actor>
(And <constraint.expression>)* # Zero or more local postconditions
# Zero or more constraints depending on the properties that are initialised
(And the property <property> of <domainEntity> is <operator> <value>) *
```

```
@Intent Create
Scenario: Settle Instruction
Given Inx of type T2S Settlement Ins does not exist in T2S of type Settlement Platform.
When T2S Settle Instruction,
Then Inx exists in T2S
 And the property state of Inx is equal to Settled
# ... more Gherkin Scenarios follow.
```

Fig. 1. Main steps of AGAC and a fragment of the *Perform a settlement* AD from Figure 3

overlooked, and third, AGAC motivates requirements analysts to make their models more precise (for example, AGAC entails the modeling of inputs and outputs). Last, augmented precision helps improve the understandability of models and facilitate further automation opportunities such as change impact analysis and regression testing.

Contributions: Motivated and inspired by current practice in requirements engineering and BDD, AGAC makes three contributions: **(1) Modeling Methodology**. UML is only a notation. To be able to effectively model requirements, one needs a specific methodology tailored to the problem domain and the objectives. Such a methodology typically includes a subset of the UML and tool-supported, modeling steps. We propose a methodology to support the derivation of AC in the context of BDD and Gherkin. Our methodology enforces compliance through a profile and dedicated tool, thus ensuring that all the information required for deriving AC and test cases is available. **(2) AC templates** capturing acceptance criteria patterns that we identified based on the analysis of 841 test specifications for three projects in the financial technology (FinTech) domain. Such patterns are a basis for analysis and automation. **(3) Algorithms and tool for automated AC generation.** The inputs to the AC generation process are requirements models and AC templates. The algorithms process the inputs, create test models, identify test model paths corresponding to valid acceptance testing scenarios, and translate these paths into sequences of Gherkin scenarios using AC templates. **(4) Industrial Case Study.** We apply AGAC to a representative financial system, to demonstrate its feasibility and benefits in a realistic context.

AGAC targets system requirements modeled using *Activity Diagrams* (ADs), *Class Diagrams* (CDs) and *Use Case Diagrams* (UCDs) [3]. ADs specify the SUT behaviors and CDs describe the types and properties of the data or objects manipulated in the ADs by actors defined in the UCDs. In our case, ADs, CDs and UCDs are the most suitable notations since they are commonly and increasingly used in Software Requirements Analysis (SRA), for example in financial institutions such as our industry partner, Clearstream Services SA Luxembourg [4]. Moreover, it is common to see large financial institutions use these diagrams to communicate with other parties. For example, the European Central Bank (ECB) uses ADs to describe parts of the pan-European platform for securities settlement, called Target2Securities (T2S) [5].

The rest of this paper is organized as follows. Section II provides an overview of AGAC. Section III describes our modeling methodology. Section IV presents some simplified versions of the diagrams in our industrial case study. Section V elaborates our AC generation procedures and templates. Section VI reports the results of our case study. Section VII reflects on the lessons learned from the case study. Section VIII discusses related work. Section IX concludes the paper.

II. Approach Overview

AGAC consists of two steps, *Create Specifications* and *Derive Acceptance Criteria*, depicted in Figure 1 along with their inputs and outputs.

The first step, *Create Specifications*, is performed manually by the engineers and consists of producing requirements models by following AGAC. These requirements models include ADs, CDs and UCDs. *Actors* are defined in UCDs (e.g., *Settlement Platform* in Figure 1) and execute *Actions* that are part of the *Activities* represented in ADs. *Domain Entities* and their properties are characterized by a *Domain Model*, represented using CDs and referenced within the ADs. For example, all the types and actors appearing in the AD in Figure 1 are specified in the CD and UCD shown in the same figure. In Figure 1, according to UML notation, the activity partition named *T2S* : *Settlement Platform* contains the actions performed by a *Settlement Platform* actor named *T2S*. The action *Settle Instruction* creates the object *Inx* of type *T2S Settlement Ins*, which is specified in the domain model. Also, after *T2S* executes the action *Settle Instruction*, the object *Inx* is expected to have its *State* set to *Settled*, one of the possible values according to the domain model.

The second step, *Derive Acceptance Criteria*, is automated and consists of matching the requirements models created by the first step with a set of predefined AC templates to generate Gherkin AC based on the intent of actions. The templates define fixed parts of the text in the Given-When-Then structure of a Gherkin scenario and leave some variable parts that will be replaced with text generated by AGAC. For example, the

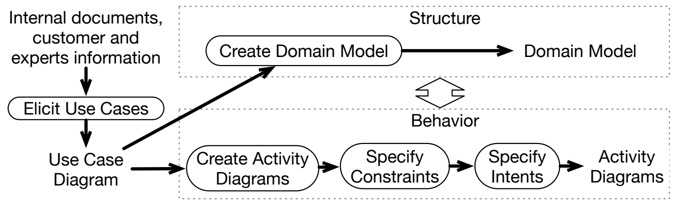


Fig. 2. Overview of the "Create Specifications" step

variable parts <actor> and <action> in Box 2 of Figure 1 are replaced with the fragments *T2S* and *Settle Instruction* to generate the acceptance criterion shown in Box 3 of Figure 1. The resulting acceptance criterion means that when the settlement platform *T2S* executes *Settle Instruction*, if the domain entity instance *Inx* does not exist, then *Inx* is created and its *State* is *Settled*.

What enables the identification of the template to be used to derive the AC for a particular action is the tag @Intent, which is specified in every AC template. Two key characteristics of our approach are the capability to automatically determine the intent of certain actions (e.g., creating an object in the case of *Settle Instruction* in Figure 1) and to enable engineers to clarify an intent when it is not possible to automatically derive it. The following sections describe in detail these two steps.

III. Create Specifications

Figure 2 outlines our methodology for creating system specifications. Our methodology has been inspired by object-oriented analysis (OOA) [6] and current practice adopted by our industry partner; it includes three widely adopted OOA activities (i.e., *Elicit Use Cases*, *Create Domain Model* and *Create Activity Diagrams*) and two additional activities (i.e., *Specify Constraints* and *Specify Intents*) that enable the automated generation of AC. In the following, we provide an overview of the three standard OOA practices adopted by our methodology and provide a detailed description of the two activities that are specific to our methodology.

According to standard OOA practice, use cases are elicited based on different sources of information such as internal documents, and information from customers and experts. Usually one UCD describes the main system Use Cases and Actors. Moreover, the analysts create at least one Activity for each Use Case, and represent the process of each Activity using at least one AD. ADs are divided in *Activity Partitions* (modeled by means of boxes with a title) and each Activity Partition is associated to an Actor. The actor associated to an activity partition is the one who is supposed to perform the activities it includes. A Domain Model is used to capture the types of Domain Entities (i.e., data or objects) handled by the activities.

Figure 3, shows example diagrams produced according to our methodology, based on a large-scale case study from the financial domain provided by our industrial partner. Given the use case *Perform a settlement*, the analyst creates an AD named *Perform a settlement* and populates the Domain Model with all the domain entities produced, modified or verified in the AD. In Figure 3, the AD *Perform a settlement* includes two activity partitions P and $T2S$, the former being executed by the actor *Participant*, the latter by the actor *Settlement Platform*. The association between activity partitions and actors is captured in the title box of the activity partition.

A. Specify Constraints

Constraints are defined as Boolean expressions used to describe pre- and postconditions of actions and activities, and guards on control flows to restrict the execution of actions. Constraints can be shown as a note box linked to any node or edge of a diagram, or in a constraints compartment. Constraints linked to Actions are called *local* pre- and postconditions. Figure 3 includes an example of a precondition for the Activity *Perform a settlement* in its constraints compartment. The expression of that precondition is $Pre : SettlementPlatform.allInstances() \rightarrow forAll(t \mid t.isInitialised = true)$. A note box is instead used to show the local postcondition *Lp1* linked to the action *Settle Instruction*.

Constraints play a crucial role in our approach since they enable the identification of the Given (i.e., the test case precondition) and Then (i.e., expected postcondition to be verified during testing) elements for Gherkin scenarios. Our methodology does *not* require all the constraints to be specified manually, sparing the engineers the need to specify implicit constraints that can be identified by analyzing the diagrams.

Implicit constraints are derived according to the following rules. If an action updates the attributes of a domain entity, then our toolset derives a precondition indicating that the domain entity instance is expected to exist. This is the case for the action *Settle Instruction* in Figure 1, which updates the state of the object *Inx* and thus implies that *Inx* exists (otherwise the action would not be able to change its state). If an action follows a condition node (represented by diamonds in the AD) then our toolset derives a precondition that matches the expression satisfying the condition node. If an action produces an object (represented by a squared box in the diagram) and one or more attribute values are specified for that object, then our toolset derives a postcondition from the specified attribute values. This is the case for *Settle Instruction* in Figure 1, for which we derive a postcondition indicating that the property *State* of *Inx* is expected to be equal to *Settled*.

B. Specify Intents

To enable the automated generation of AC, our methodology requires that analysts specify the intent of actions by relying on a set of predefined stereotypes. Figure 4 shows the *Intents* profile provided by AGAC. Intents capture the observable behavior of an action that should be verified during testing. The stereotypes *Create*, *Read*, *Update* and *Send* indicate that the domain entity connected to the outgoing edge of the action has been created, read, updated or sent by the action. The stereotypes *Delete*, *Receive* and *Validate*, instead, indicate that the domain entity connected to the incoming edge has been deleted, received or validated by the action,

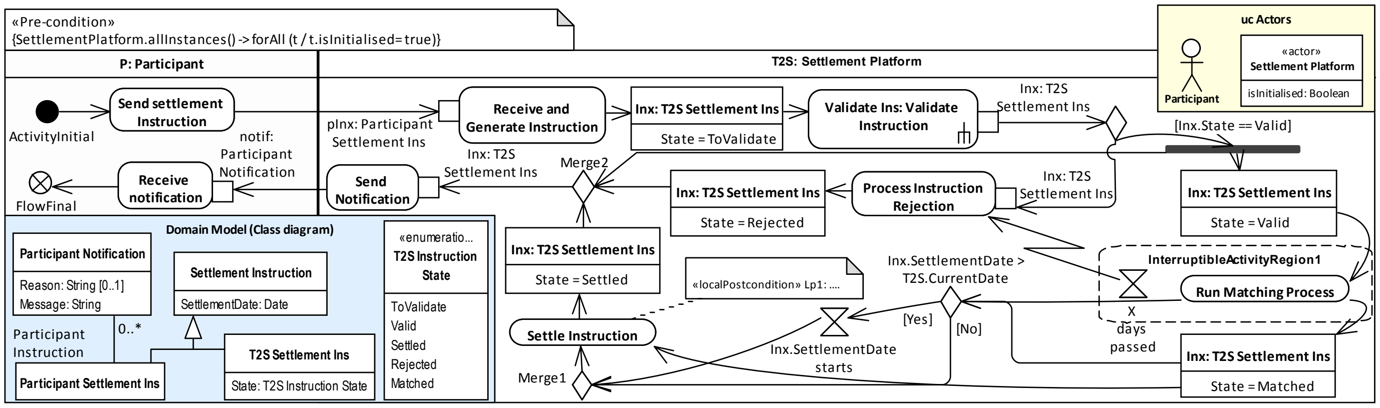


Fig. 3. Simplified diagrams describing how to perform a settlement

respectively. The stereotypes $Display$, $Enable$, $Disable$, and $NotDisplay$ indicate that the action affects the user interface. More specifically, $Display$ and $NotDisplay$ indicate that the domain entity connected to the outgoing edge of the action is expected to be displayed or hidden from the GUI, respectively. $Enable$ and $Disable$ indicate that the GUI enables or prevents an actor from accessing or modifying the domain entity.

Similarly to constraints, intents may not be explicitly declared if they can be automatically derived. This occurs in the following cases. (Case-1) The $Update$ stereotype is automatically assigned to an action when the input and output edges of the action are connected to domain entities with the same identifier. This is the case for the action $Run\ Matching\ Process$ in Figure 1 which is expected to update the $State$ property of the domain entity instance named *Inx*. (Case-2) The *Create* stereotype is assigned to an action when the output edge of an action is connected to a domain entity with an identifier that is not encountered when processing previous actions. (Case-3) Any stereotype can be automatically assigned to an action if the name of the stereotype matches the verb appearing in the action's name, or one of its synonyms (e.g., $Receive\ Instruction$ and $Validate\ Instruction$ match the stereotypes $Receive$ and $Validate$).

To support engineers in the definition of ADs, the AGAC toolset displays warnings when inconsistencies are encountered in the stereotypes, the verbs appearing in the action names, or the inputs and outputs of the actions, based on the derivation rules described above. For example, AGAC would display a warning if an action does not create any output data, but (1) its main verb is *create* or a synonym (e.g., produce, generate), or (2) the action has the $Create$ stereotype.

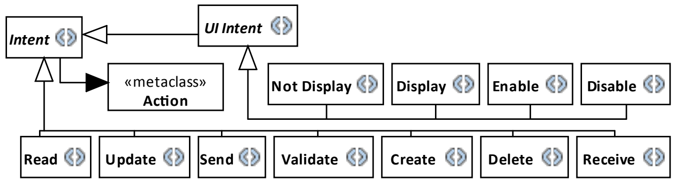


Fig. 4. Intents profile that makes explicit the intent of actions in ADs

IV. Modeling Example

In the following, we provide additional details about the example in Figure 3, which is required to exemplify the *Derive Acceptance Criteria* step of AGAC (Section V). The AD starts at the $InitialNode$ (a black circle) followed by the action called $Send\ Settlement\ Instruction$ (the rounded rectangle) performed by a $Participant$ (e.g., a transfer agent referred in the diagram as P). That action creates the $Participant\ Settlement\ Ins$ object $pInx$ that is received by the actor $T2S$ (Target2Securities) of type $Settlement\ Platform$. Based on $pInx$, $T2S$ creates the settlement instruction Inx that will be processed during the entire activity and whose property $State$ is initially set to $ToValidate$.

The $CallBehavior$ action $Validate\ Ins$ invokes the activity $Validate\ Instruction$ (not shown in the diagram) that makes business validations (e.g., checks for duplicates, restrictions and privileges), and updates the Inx state to $Valid$ if it passes the validation. If the state is different than $Valid$, it is reset to $Rejected$ and a notification, which includes the $Reason$ of rejection, is sent to the participant. If the instruction is $Valid$, T2S executes three actions in parallel: (1) $Send\ Notification$, which informs the participant that its instruction is valid and waits for a matching instruction to arrive before being settled. (2) $Run\ Matching\ Process$, which compares the settlement details of the settlement instructions provided by the deliverer and the receiver of securities to ensure that both parties agree on the settlement terms of the transaction. (3) Wait for the time event $X\ days\ passed$ to occur. If the matching process finishes before the time event $X\ days\ passed$ occurs, Inx will be $Matched$, otherwise, it will be $Rejected$.

If the $SettlementDate$ of Inx is in the future (see the constraint $Inx.SettlementDate > T2S.CurrentDate$), T2S will wait until that date before executing $Settle\ Instruction$ (as shown by the $AcceptTimeEventAction$ node labeled as $Inx.SettlementDate$). Finally, after the instruction is settled, the $State$ property of the Inx object will change to $Settled$ and a notification will be sent to the participant.

Figure 3 shows a CD representing the domain model (fragment) related to the AD shown in the same figure. The domain model shows that $Settlement\ Instruction$ has the property $SettlementDate$, and there are two types of

*Settlement Instruction*s, one for the participant and the other for *T2S*. A *Participant Instruction* can have zero or more *Participant Notification*s which contain a message and, in some cases, a reason as to why the instruction was rejected. The domain model also shows an enumeration of the possible states of *T2S Settlement Ins*.

V. Derive Acceptance Criteria

Figure 5 shows an overview of the *Derive Acceptance Criteria* step, which is fully automated by our toolset. This step consists of three sub-steps: *Generate Test Model(s)*, *Generate Paths* and *Generate AC*.

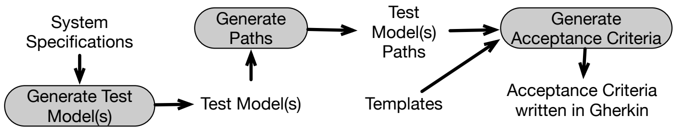


Fig. 5. Overview of the "Derive Acceptance Criteria" step

A. Test model

Test models capture, in explicit form, information that is necessary for the correct generation of acceptance testing scenarios. More precisely, a test model captures the control flow explicitly modeled by an AD and augments it with information that is implicitly modeled in the AD, thus enabling a clear identification of (1) inputs and outputs, (2) events that start the processes depicted by the AD, and (3) events that interrupt the execution of regions of the AD. These elements are not explicitly indicated in an AD (e.g., it is not possible to identify an interrupting event without analyzing its outgoing edges) but are required to enable test generation. AGAC automatically derives this information by processing the ADs produced following our methodology.

Figure 6 shows the test model metamodel used by our approach. The metamodel consists of two groups of metaclasses: metaclasses representing the elements in an AD (in gray) and metaclasses capturing the control flow information necessary for building acceptance testing scenarios.

The **structure of the AD** is captured using an ordered pair *Graph* composed of a set of *Vertices* together with a set of *Edges*. Each edge is associated with two different vertices and can have a *Label* and *Type*. Each vertex belonging to a graph represents one element in an AD (e.g., an *Action*). For completeness, an excerpt of the metamodel of ADs is available online [7]. *Vertex* is defined as a 12-tuple ⟨*Id, Name, Type, Actor, ActorType, Parent, Classifier, LocalPreconditions, LocalPostconditions, NextList, PreviousList, Properties*⟩. *Id* is a unique identification of the node, *Name* is an optional text by which a node is known, *Type* is the node's UML metaclass (e.g., *Action*), *Actor* is the role played by a system that executes the node, *Parent* is a reference to the node that contains the node in the diagram (e.g., an *ActivityPartition*), *Classifier* is a reference to the corresponding domain entity in the domain model, *LocalPre* and *LocalPost* are the local pre- and postconditions of a node, *Properties* are defined in the domain model as class properties. *NextList* and *PreviousList* contain the edges to the next and previous vertices in the AD. For example, the vertex representing the action "Settle Instruction" in Figure 3 is represented by the tuple ⟨*Id: an-id, Name: Settle Instruction, Type: Action, Actor: T2S, Parent: a reference to the Activity "Perform a settlement", Classifier: null, LocalPreconditions: null, LocalPostconditions: Lp1, NextList: [Inx], PreviousList: [Merge1, Inx], Properties: [null]*⟩.

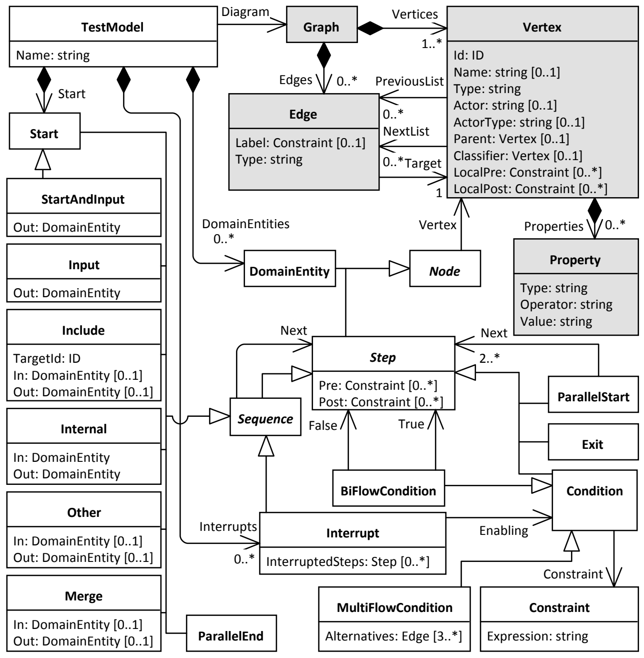


Fig. 6. Test model metamodel

The **test model control flow** is captured by *Node* classes. Each *Node* class is associated to the *Vertex* it has been derived from. Nodes can be either *DomainEntities* or *Steps*. A *Step* has zero or more pre- and postconditions. These conditions include the ones in the *LocalPre* and *LocalPost* attributes of the associated vertex but are not limited to them since our toolset may automatically derive pre- and postconditions as indicated in Section III-A. A *Step* refers to four types of nodes that differ regarding the number of following steps they are connected to: (1) *Sequence* has a single next step; (2) *Exit* does not have any next step; (3) *ParallelStart* has two or more next steps that will be executed concurrently; (4) *Condition* has two or more possible next steps and a reference to a *Constraint*. Among the subtypes of *Condition*, a *BiFlowCondition* has two next steps (*True* and *False*), while a *MultiFlowCondition* is associated to three or more alternative next steps.

There are eight types of *Sequence* nodes: *Start*, *StartAndInput*, *Input*, *Internal*, *Interrupt*, *Include*, *ParallelEnd* and *Other*. *Start* and *StartAndInput* represent the beginning of an activity. *Input* indicates that the acceptance testing scenario should include an operation that provides a *DomainEntity* to the system. *Internal* indicates that the system alters its internal state. The effects of an internal step are specified as postconditions (see the *Post* property in *Step*). *Interrupt* indicates the presence of input

TABLE I
TRANSFORMATION RULES USED BY AGAC FOR CREATING TEST MODEL NODES FROM AN AD.

Generated Test Model Node	Transformation Rule
Input	The type of the vertex is not *ControlNode* and (1) the vertex has one output and no input domain entities or (2) the vertex has one input and one output domain entity with different names.
Internal	The type of the vertex is not *ControlNode* and the vertex has one input and one output domain entity with the same name.
Start	The type of the vertex is (1) *InitialNode*, or (2) an *AcceptEventAction* or *AcceptTimeEventAction* that does not have incoming edges and is not an *Interrupt*.
Exit	The type of the vertex is *ActivityFinal* or *FlowFinal*.
Include	The type of the vertex is *CallBehaviourAction*.
Interrupt	The type of the vertex is *AcceptEventAction* or *AcceptTimeEventAction*, the vertex belongs to an *InterruptibleActivityRegion*, and is connected to an *InterruptFlow* exiting from that region.
BiFlowCondition	The type of the vertex is *Decision* and the vertex has two outgoing edges.
MultiFlowCondition	The type of the vertex is *Decision* and the vertex has three or more outgoing edges.
DomainEntity	The type of the vertex is an object node such as *InputPin*, *OutputPin* or *ActivityParameterNode*.
ParallelEnd	The type of the vertex is *Join*.
ParallelStart	The type of the vertex is *Fork*.
Other	The vertex does not match any of the rules above.

events that enable the termination of the activities belonging to the steps listed in the property *InterruptedSteps*, if the associated *Condition* evaluates to true. *Include* invokes the test model associated to the activity specified in the *TargetId* property. *ParallelEnd* joins two or more steps that run concurrently. *Other* is a node that has a next step but was not recognized as one of the other seven types of *Sequence*.

B. Generating Test Model(s) from AD

We generate a test model by performing a depth first traversal of the AD that visits all the vertices that are connected to the root of the AD. For each vertex visited we generate a corresponding *Node* in the test model (i.e., an instance of the metaclass *Step* or *DomainEntity*). To determine the type of *Node* to be created for a vertex in the AD, we defined a set of transformation rules that take into account the type of the *Vertex* and its connections to other vertices. The transformation rules are summarized in Table I, with a more detailed description of each rule provided in the following paragraphs. In our implementation, to perform the traversal of the AD, we rely on the C# UML API of Enterprise Architect [8]. Figure 7 represents the test model created for the example in Figure 3 using a UML object diagram.

Input. A vertex represents an *Input* node if it is not an AD *ControlNode* (e.g., *Fork*, *Decision*, *Merge*, *Join*) and one of the two following cases hold: (1) The vertex has one output and no input domain entities, which indicates that the vertex provides a new domain entity to the system. (2) The vertex has one input and one output domain entity, but they are different, thus indicating that, after receiving some input entity, the vertex provides a new domain entity to the system.

Internal. A vertex represents an *Internal* node if it is not an AD *ControlNode* and has one input and one output domain entity with the same name, which indicates that the vertex performs some processing on the given domain entity.

Start and *StartAndInput*. A *Start* node is created for every vertex of type *InitialNode* in an AD. A *StartAndInput* node is created for every *AcceptEventAction* and *AcceptTimeEventAction* (e.g., *X days passed* in Figure 3), which are the vertices that start flows in an AD after an event is received or after a given timer expires, respectively. For both *Start* and *StartAndInput*, we add as preconditions all the local preconditions of the corresponding vertex in the AD and the preconditions of the parent vertices. For example, the initial activity in Figure 3 does not have any local precondition, but its container, the ***Perform a settlement*** Activity, has the precondition starting by $Pre : SettlementPlatform.allInstances() \ldots$ We associate this precondition to the *Start* node, ***ActivityInitial***. In general, for every created *Node*, we add the local pre- and postcondition connected to the corresponding vertices.

DomainEntity, *Exit*, *ParallelStart*, *ParallelEnd*, *Include*. These nodes are created when the vertex's type matches one of those indicated in the corresponding lines of Table I.

Condition nodes that are generated from vertices of type *Decision*. The condition subtype (i.e., *BiFlowCondition* or *MultiFlowCondition*) depends on the number of outgoing edges; more precisely, a *MultiFlowCondition* is created if the number of outgoing edges is greater than two. The *Expression* of the *Constraint* associated to the condition node is based on the label of the vertex. In the case of *BiFlowCondition*, if the vertex has a name, we use this name as the expression (Boolean expressions are often written within the diamonds in ADs). If the vertex does not have a name, we use the label associated to one of the outgoing edges. For nodes of type *MultiFlowCondition*, we rely on the labels appearing on each outgoing edge (if a label is missing, AGAC reports a warning).

Interrupt. An interrupt is caused by the presence of input events that enable the evaluation of a condition that may terminate the execution of the steps. More precisely, in an AD, an *InterruptibleActivityRegion* is represented as a dotted rectangle that surrounds a group of Activity elements, all affected by certain interrupting events (i.e., *AcceptEventAction* or *AcceptTimeEventAction*). Any activity occurring within the bounds of an Interruptible Activity Region is terminated when an execution flow originating from an interrupting event belonging to the region exits from the region through an *InterruptFlow* (represented using a zig-zag connector). This is the case of the Interruptible Activity Region in Figure 3, which is interrupted when the *AcceptTimeEventAction* *X days passed* is triggered and *InterruptFlow* reaches the action *Process Instruction Rejection*.

Interrupt nodes cannot be directly identified from a specific set of metaclasses because *AcceptEventAction* and *AcceptTimeEventAction* are used also to model the beginning of normal processing flows. AGAC identifies the *AcceptEventAction* or *AcceptTimeEventAction* vertices that are connected to an *InterruptFlow*, itself exiting from an Interruptible Activity Region, to determine if a vertex represents an *Interrupt*.

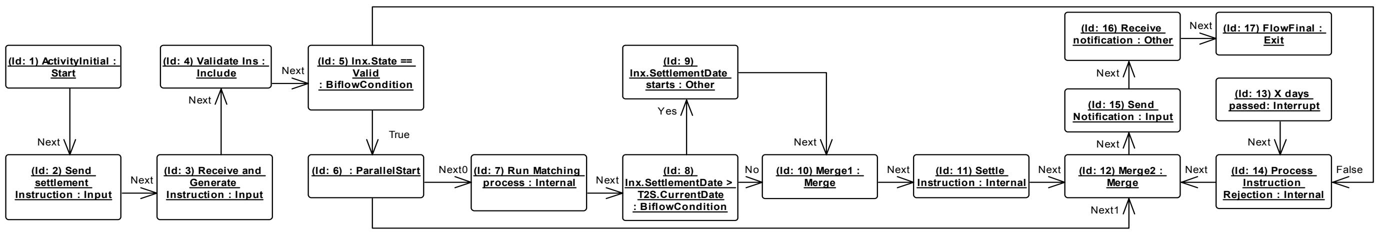

(Id: 1) ActivityInitial : Start
Next
(Id: 2) Send settlement Instruction : Input
Next
(Id: 3) Receive and Generate Instruction : Input
Next
(Id: 4) Validate Ins : Include
Next
(Id: 5) Inx.State == Valid : BiflowCondition
True
(Id: 6) : ParallelStart
Next0
(Id: 7) Run Matching process : Internal
Next
(Id: 8) Inx.SettlementDate > T2S.CurrentDate : BiflowCondition
Yes
(Id: 9) Inx.SettlementDate starts : Other
No
Next
(Id: 10) Merge1 : Merge
Next
(Id: 11) Settle Instruction : Internal
Next
(Id: 12) Merge2 : Merge
Next1
Next
(Id: 15) Send Notification : Input
Next
(Id: 16) Receive notification : Other
Next
(Id: 17) FlowFinal : Exit
(Id: 13) X days passed: Interrupt
Next
(Id: 14) Process Instruction Rejection : Internal
Next
False

Fig. 7. Test model for the AD in Figure 3. Nodes' properties have been hidden to simplify reading. The numbers in brackets (Id:n) show the identifiers referred to in the paper.

C. Generation of test paths

Test models can be traversed as directed graphs and we use the term *test path* to refer to a traversal of the test model. A *path* is a sequence $[n_1, n_2, ...n_M]$ of nodes connected by an edge. A test path p shows the order of actions executed during a complete execution of the process described by an AD. p is said to *visit* node n if n is in p.

In our approach, each test path starts in a *Start*, *StartAndInput* or *Interrupt* node and ends in an *Exit* node. The types of nodes found in test paths are *Steps*, while the types of edges connecting the nodes are the valid relationships between the Steps, i.e., *Next*, *True*, *False*, or $Next_i$ where i represents a thread i originating from a *ParallelStart* node.

AGAC uses standard depth- and breadth-first search algorithms [9] to traverse non-concurrent and concurrent nodes, respectively. These algorithms are used by prior works in graph-based generation of test cases (some example approaches can be seen in a survey made by Shirole and Kumar [10]).

In AGAC, the breadth-first traversal of the graph G starts at the *ParallelStart* node and explores all of the immediately next nodes at the present depth prior to proceeding with the nodes at the next depth level. The breadth-first traversal enables the maximization of the number of threads interleavings by exercising, in sequence, actions that belong to different threads. This is a common heuristic to find deadlocks or race conditions that cause null pointer exceptions or assertion violations [11]. In addition, AGAC enables engineers to specify the maximum number of times a node belonging to a loop should be visited. More details about how AGAC traverses concurrent nodes are available online [7].

AGAC produces four valid test paths for the test model in Figure 7: p_1=$[1, 2, 3, 4, 5, 6, 12, 7, 15, 8, 16, 9, 17, 10, 11, 12, 15, 16, 17]$ (executed when $Inx.State = Valid$ and $Inx.SettlementDate > T2S.CurrentDate$); p_2=$[1, 2, 3, 4, 5, 6, 12, 7, 15, 8, 16, 10, 17, 11, 12, 15, 16, 17]$ (executed when $Inx.State = Valid$ and $\neg(Inx.SettlementDate > T2S.CurrentDate)$; p_3=$[1, 2, 3, 4, 5, 14, 12, 15, 16, 17]$ (executed when $Inx.State \neq Valid$); and p_4=$[13, 14, 12, 15, 16, 17]$ (executed after $X\ days\ passed$).

D. Generation of Acceptance Criteria

AGAC generates acceptance criteria (AC) based on a predefined set of templates derived from a study of 841 tests specifications created by our industry partner for three recent and representative projects, involving three distinct testing teams. In comparison to the AC generated by AGAC, the 841 test specifications written by the testers included many implementation details and were not derived from any requirement model. We analyzed and identified the test intents of each test specification and created a template for each intent type. We found eight intent types and their distribution is shown in Figure 8. `Update` was the most frequent intent, counting for almost half of the total. We added the `Delete`, `Interrupt` and `Disable` to the set of intent types because they were not found in the 841 test cases, but our partners deemed them necessary, though infrequent.

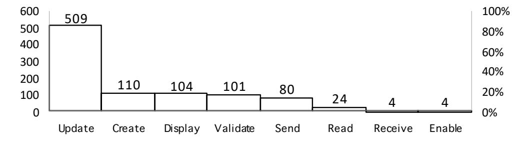


Fig. 8. Distribution of each intent type for the 841 test specifications

The bottom rows of Figure 9 show the templates, while the uppermost rows specify the keywords used in each template. In the following, we describe the testing activities performed by every template and provide additional information on how each of these activities can be automated in the generated test cases.

`Create` verifies that an entry (i.e., an instance of a domain entity) does not exist before the execution of the action and that the entry is created after the execution of the action. `Delete` verifies that a certain entry existed in the system before the execution of the action but no longer exists after its execution. `Send` verifies that a certain entry exists in the system before the execution of the action and has been sent after its execution. `Receive` verifies that (1) an entry does not exist in the system before the execution of the action, (2) the entry has been received, and (3) the entry exists after the execution of the action. `Read` verifies that an entry exists in the system and the actor reads it (usually after a request for retrieving, searching or visualizing data). `Update` verifies that an entry exists in the system, that the actor reads it, and that certain properties (i.e., the postcondition in the test model) hold after the execution of the action. `Interrupt` verifies that an action belonging to the interruptible set of actions is running on the system and then, after the interrupting action is executed (i.e., a given clock has been triggered or an event has been received), all the actions belonging to the interruptible set of actions have been terminated. `Validate` verifies that a certain postcondition holds after the execution of an

Acceptance_ Criterion_File	FEATURE_NAME # The AD name. BACKGROUND ? # Common preconditions SCENARIO_BLOCK + # One or more scenario blocks
FEATURE_NAME	Feature : <Activity>
BACKGROUND	Background : Given <FREE_EXPR> # First Activity precondition (And FREE_EXPR)* # Zero or more preconditions from the Activity
SCENARIO_BLOCK (a.k.a, AC_RULE)	(CREATE | READ | UPDATE | DELETE |SEND | RECEIVE | ENABLE | DISABLE | DISPLAY | NOTDISPLAY | INTERRUPT) (And FREE_EXPR)* # Zero or more extra local-postconditions
EXPRESSION	PROP_EXPR | EXISTS | N_EXISTS | FREE_EXPR
FREE_EXPR	TEXT # Any text that expresses a constraint
SCENARIO_NAME	Scenario: <action>
PROP_EXPR	the property <property> of <domainEntity> is <operator> <value>
N_EXISTS	<domainEntity> (of type <class1>)? does not exist in <actor> (of type <class2>)?
EXISTS	<domainEntity> (of type <class1>)? exist in <actor> (of type <class2>)?
WHEN	When <actor> <action>,
CREATE	@Intent: Create # Tag SCENARIO_NAME Given N_EXISTS (And FREE_EXPR) * # Previous conditions and local preconditions WHEN Then EXISTS (And PROP_EXPR) * # Optionally, the initialised properties
READ	@Intent: Read # Tag SCENARIO_NAME Given EXISTS (And FREE_EXPR) * # Previous conditions and local preconditions WHEN Then <actor> reads <domainEntity>
UPDATE	@Intent: Update # Tag SCENARIO_NAME Given EXISTS (And FREE_EXPR) * # Previous conditions and local preconditions (And PROP_EXPR) + # The properties that will be updated WHEN Then <actor> reads <domainEntity> from <actor> (And PROP_EXPR) + # The updated properties
DELETE	@Intent: Delete # Tag SCENARIO_NAME Given EXISTS (And FREE_EXPR) * # Previous conditions and local preconditions WHEN Then <domainEntity> does not exists in <actor>
SEND	@Intent: Send # Tag SCENARIO_NAME Given EXISTS (And FREE_EXPR) * # Previous conditions and local preconditions (And PROP_EXPR) * WHEN Then <actor> sent <domainEntity>
RECEIVE	@Intent: Receive # Tag SCENARIO_NAME Given N_EXISTS (And FREE_EXPR) * # Previous conditions and local preconditions (And PROP_EXPR) * WHEN Then <domainEntity> was received and exists in <actor>
INTERRUPT	@Intent: Interrupt # Tag SCENARIO_NAME Given <action> is running in <actor># First interruptible step (And <action> is running in <actor>) * # Other interruptible steps (And FREE_EXPR) * # Previous conditions and local preconditions (And PROP_EXPR) * When the event "<event>" happens in <actor>, Then <action> is interrupted in <actor> # First step to be interrupted (And <action> is interrupted in <actor>) * # Other steps to be interrupted
VALIDATE	@Intent: Validate # Tag SCENARIO_NAME ((Given FREE_EXPR) (And FREE_EXPR) *)? # Previous conditions and local preconditions WHEN Then FREE_EXPR (And FREE_EXPR) * # Post. related to validate
DISPLAY / NOT DISPLAY	@Intent: Display # This tag can also be "@Intent: Not Display" SCENARIO_NAME # There is no Given template for Display and Not Display (Given FREE_EXPR) * # Previous conditions and local preconditions WHEN Then <domainEntity> is displayed in the <actor> UI # Or "is not displayed"
ENABLE / DISABLE	@Intent: Enable # This tag can also be "@Intent: DISABLE" SCENARIO_NAME Given EXISTS # Exists is used for DISABLE and ENABLE (And FREE_EXPR) * # Previous conditions and local preconditions WHEN Then <domainEntity> is enabled in the <actor> UI # Or "is disabled"

Fig. 9. Templates based on acceptance criteria types

action. `Display (Not Display)` verifies that a certain entry is displayed (not displayed) on the user interface after the execution of an action. `Enable (Disable)` verifies that, upon the execution of an action, a given user-interface widget (e.g., a button or a text box) becomes enabled / modifiable (alternatively, disabled / unmodifiable). From these short descriptions, we can see that these AC types are likely to generalize to many other information systems.

In AGAC, each valid test path produces one AC. Nodes in the test path will produce "Given-When-Then" structures (i.e., Gherkin scenarios). Guard expressions of the edges between the nodes will produce preconditions that will complement the local preconditions of the node.

A template is instantiated by replacing the parameters of the template with the corresponding properties extracted from the test model node. The selection of the template depends on the stereotypes applied to the vertex associated to a node (see Section III-B). AGAC can automatically assign a stereotype to most of the nodes. *Input* and *StartAndInput* nodes are assigned the stereotype *Create* because they are expected to introduce new data into the system. *Internal* nodes are assigned the stereotype *Update* since they modify the system state. *Start* and *Exit* nodes are not assigned any stereotype because the former leads to the instantiation of the *Background* template while the latter leads to the verification of the AD's postconditions. The cases where a stereotype (e.g., *Display*, *Not Display*, *Enable* and *Disable*) needs to be specified manually are for the nodes classified as *Other*. The AC generated for the test model shown in Figure 7 are available online [7].

VI. Evaluation

We have performed an empirical evaluation of AGAC aimed at addressing the following research questions:

RQ1. Is AGAC feasible to apply in a real setting? We elaborate this question into two sub-questions: **RQ1.1.** Can practitioners conveniently model all the requirements using our methodology? **RQ1.2.** Can our automated algorithms analyze the resulting requirements models and generate appropriate test specifications in the Gherkin language?

RQ2. Is the additional effort associated with using AGAC justified by its benefits?

A. Object of the study

We perform our evaluation in the context of an industrial case study. The object of the study is an online banking system being developed by our collaborating partner. This system provides a suite of investment fund services with a single point of entry and a standardized procedure for transactions related to more than 190K investment funds around the world. The system is representative of the large majority of the IT systems developed by our partner. A team of four analysts (three senior and one junior) at the partner followed our proposed methodology to build a complete set of functional requirements models for the system. The team had already been trained in the methodology prior to modeling using model examples such as the one shown in Figure 3. The training took about 1 hour and was focused mostly on how to model object flows in ADs. The researchers were not involved in model construction, supporting the practitioners only by answering their questions about the methodology and reviewing the models to ensure the methodology had been followed correctly. The resulting models are composed of: (i) a domain model with 176 elements, an element being a class, enumeration, property, association or generalization, (ii) 14 activities, each of which is elaborated using an individual AD. Collectively, the 14 ADs contain 434 elements, an element being an activity parameter, action, stereotype, pin, object, decision/synchronization/merge node, partition or interruptible region.

B. Methods and results

RQ1.1. The practitioners were able to successfully apply the methodology for modeling the requirements of the system in our study. All the team members participated in the modeling of the 14 ADs. While two members focused on creating the models, the others discussed and suggested changes. The modeling activities were completed within the time budget they had set aside for the task (over one month, 2 hours per week).

Based on the feedback we received from the practitioners and our own review of the models, no major issues were found that would impede the adoption of the methodology. Nevertheless, we note that turning AC into executable test cases may require formal constraints, which can be automatically derived from textual representations using dedicated natural language processing solutions [12].

RQ1.2. We applied the AC generation process of Section V-D to the 14 ADs in our case study. Our algorithm successfully handled all the models and generated a total of 137 AC. The minimum and maximum number of Gherkin scenarios per acceptance criterion is 1 and 13, respectively. In total, the 137 AC include 990 Gherkin scenarios, yielding an average of 7 scenarios per acceptance criterion. The AC generation process further confirmed the effectiveness of our strategy for the automated identification of intents. Specifically, out of the 990 Gherkin scenarios, 977 were derived from intents that were identified automatically. The execution time of AC generation was negligible: all the 137 AC were generated in 25 seconds on a computer with an Intel i7 CPU and 4GB of memory.

We verified the quality of the 137 AC by randomly selecting 14 (10%) of them to be validated with the experts in our case study. The experts confirmed that all the AC were understandable and correct. By correctness, we mean that the tool accurately inferred the intents types of all the actions in the AC, all the expected scenarios were covered, and all the pre- and post-conditions were translated from the models to the `Given` and `Then` blocks of individual Gherkin scenarios.

RQ2. Our proposed methodology requires information that is additional to what our partner's current modeling practice would capture. It is thus important to examine whether the benefits that our approach brings outweigh the modeling overhead. We argue about the benefits of our approach indirectly by measuring the number of additional elements that our methodology requires. Upon investigating our case study models, we determined that out of the total of 610 elements in these diagrams, 461 (76%) can be traced back to the existing modeling practice at the partner. Only 149 model elements (24%) are imposed by our methodology.

In return for the extra modeling effort, and noting that the overall effort is still within an acceptable range (see RQ1.1), our approach provides three key benefits. First, it eliminates the need to specify and revise AC manually. This is particularly important in an agile setting, where requirements change frequently. While we do not have a detailed effort breakdown for the different requirements analysis tasks performed by our partner, the experts in our study estimated that as much as 50% of the entire requirements analysis effort in any given project would go to creating and updating the AC. Considering that the generated AC include 990 Gherkin scenarios, they found automated AC generation to present a major cost advantage. A second important benefit of our approach is that the AC are now produced systematically, thus decreasing the likelihood that critical system-level scenarios would be overlooked. A final benefit of our approach has to do with making the models more precise. Doing so not only helps increase the understandability of the models but also increases automation opportunities for other challenging tasks such as change impact analysis and regression testing. Although user studies are required, our case study results strongly suggest that the benefits of our approach justify its modeling overhead.

We also probed the opinion of practitioners regarding the AGAC tool support. One main factor facilitating the adoption of the tool was its tight integration with a modeling environment that is familiar to them (i.e., Enterprise Architect [8]).

The future of AGAC is promising within the industry partner organization. There are groups related to DevOps and functional analysis that are interested in AGAC. Also, there is already a support service provider that will provide long-term technical support for AGAC, thus making the adoption and evolution of AGAC possible on the long term in the production environment.

C. Threats to validity

Internal validity. Internal validity concerns how well an empirical study is performed, particularly in terms of avoiding confounding factors. The main confounding factor to mitigate in our case study is the influence of the researchers on the modeling activities. Due to our proposed methodology being new to the collaborating partner, we inevitably had to mentor the practitioners to ensure that the models were built correctly. Nevertheless, our level of involvement was not more than what one would expect from an expert consultant. All the modeling activities were led by the practitioners, with the researchers' role being limited to question answering and providing comments on the models built. Assuming that the practitioners are adequately trained, we do not believe that the researchers' involvement in the case study influenced the empirical observations.

External validity. External validity concerns the generalizability of our results. Although our empirical evaluation is based on a single case study, the system in this case study is representative of a broader class of business-critical systems, namely banking and securities systems providing services through the Web. We thus anticipate that the observed results will generalize to other systems of the same nature. Additional case studies involving other types of systems that are commonly modeled by means of ADs, e.g., public administration services, are nevertheless necessary to improve external validity.

VII. Lessons Learned

The application of AGAC on pilot projects led to a number of lessons learned that are summarized below.

AC are a strong motivator for building better models. Developers may question the cost of applying a systematic modeling methodology. The ability to automatically generate AC serves as a compelling motivation for building better models, due to the immediate cost reductions brought about by AC automation. The resulting models, in addition to supporting AC generation, have additional benefits such as: traceability between requirements and test cases, improved understandability and better maintainability.

Using a template format for AC is beneficial. Our partners did not use predefined templates and this led to vagueness and ambiguity. Experts saw the use of our Gherkin templates as a good strategy for mitigating the plethora of potential issues that can arise when unrestricted natural language is used for writing test specifications. Furthermore, the templates provide a direct path to executable test cases via the extended facilities and infrastructure around Gherkin (e.g., Cucumber [2]).

AC provide a feedback loop for detecting incompleteness in models and incrementally improving the models. Practitioners can quickly generate AC and review them to verify that all the information is complete in the AC. If there is any incomplete information (e.g., the Actor name is missing in a `When` block in Gherkin), the practitioners can complete their requirements models and regenerate the AC. This cycle of generating AC, reviewing the results, and addressing inaccuracies in models, enables incremental improvement.

Analysts and testers unlikely generate system-level AC systematically. Generating system-level AC involves systematically exercising different model paths while trying to maximize certain properties such as interleaving, coverage, etc. This activity is cognitively very difficult for humans due to the very large number of possible paths. Going from ad-hoc to systematic system-level AC necessarily involves automation. Our approach provides a direct solution to this problem.

Engineers need a dedicated language for constraints. We observed that the practitioners in our case study were not proficient at specifying formal constraints (e.g., OCL) beyond those that could be expressed using property/operator values and simple arithmetic. Since the target language for the AC is textual, the analysts had the flexibility to specify the more complex constraints in natural language – what mattered here was for the constraints to be explicit; the exact representation does not play a role if the constraints are understandable by the engineers and at the same time structured enough to be transformed into precise formulae, e.g., OCL, that can be verified over the models (e.g., the work of Wang et al. [13] about OCL constraints generation in use case specifications).

VIII. Related Work

The benefits of automatic generation of test artifacts are widely acknowledged today [14], [15]. There are two main types of research proposed for generating test cases from requirements specifications. The first one relies on models [10] and the second one on textual descriptions [16]. Model-based approaches [10], [17] differ in terms of the model types they process and generate. Approaches targeting UML activity diagrams aim at identifying sequences of actions that satisfy certain properties, such as guaranteeing some form of structural coverage [18], [19] or exercising concurrent behaviors [20], [21]. Approaches that generate test cases directly from textual descriptions [16], [22]–[25] seek to minimize the use of models, by relying mostly on requirements specifications in natural language. For example, UMTG [22] generates executable system test cases from use case specifications and a domain model. These approaches cannot be adopted in contexts where requirements capture the characteristics of business processes and are represented using ADs.

In general, the main factors that distinguish different approaches to requirements-driven test generation are (1) the form and content of the requirements specifications, and (2) the target representation for the tests. In the case of AGAC, the former is tailored to complex processes expressed as ADs, and the latter is targeting a well-known agile methodology (BDD) and the Gherkin language. While our work is grounded in the financial domain, we note that BDD is a common practice in many sectors, and that the AC types we have identified are expected to generalize to many other domains.

Furthermore, AGAC provides a modeling methodology aimed at ensuring that the requirements specifications contain the information both required by analysts and for generating AC. Finally, the approach takes advantage of lightweight natural language processing (i.e., generation of stereotypes based on action names) in order to simplify modeling activities and reduce their associated costs.

IX. Conclusions

In this paper, we bridge the gap between requirements models and acceptance criteria (AC) in the context of agile development and more specifically, Behavior-Driven Development (BDD). We provide a UML-based modeling methodology and an automated solution to generate AC in the form of sequences of Gherkin scenarios, Gherkin being a common language in the context of BDD. The resulting AC can subsequently be used for generating executable test scenarios.

Our modeling methodology augments traditional modeling practices with the possibility to specify the testable intents of the activities in UML activity diagrams (ADs). The methodology further comes equipped with an algorithm to automatically derive the intents in most of the cases. Given a set of models built according to our methodology, our approach then automatically identifies test paths in the ADs and, relying on a set of templates derived from a study of more than 800 test specifications in the financial domain, automatically transforms the paths into AC represented in the Gherkin language.

Our empirical evaluation, which was conducted by means of an industrial case study in the financial domain, provides initial but strong evidence of the feasibility and benefits of our approach.

Acknowledgment

We acknowledge funding from Clearstream, FNR Luxembourg (grant BRIDGES18/IS/13234469/IMoReF), and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277).

References

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[2] M. Wynne and A. Hellesoy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2017.

[3] OMG, "OMG unified modeling language (OMG UML). version 2.5," Object Management Group, Inc. http://www.omg.org/spec/UML/2.5/, Tech. Rep. formal/2015-03-01, 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/

[4] Clearstream, "Clearstream services SA," https://www.clearstream.com.

[5] ECB, "Target2-Securities. User detailed functional specifications. v3.0," European Central Bank, https://www.ecb.europa.eu/paym/t2s/html/index.en.html, Tech. Rep., March 2018.

[6] C. Larman, *Applying UML and Patterns:An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall Professional, 2002.

[7] M. Alferez, F. Pastore, M. Sabetzadeh, L. C. Briand, and J.-R. Riccardi, "Bridging the gap between requirements modeling and behavior-driven development, supplementary materials," http://hdl.handle.net/10993/39710, 2019.

[8] SPARX, "Enterprise architect," https://sparxsystems.com/products/ea/.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[10] M. Shirole and R. Kumar, "UML behavioral model based test case generation: a survey," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 4, pp. 1–13, 2013. [Online]. Available: http://doi.acm.org/10.1145/2492248.2492274

[11] A. Groce and W. Visser, "Heuristic model checking for java programs," in *SPIN*, ser. Lecture Notes in Computer Science, vol. 2318. Springer, 2002, pp. 242–245.

[12] C. Wang, F. Pastore, and L. Briand, "Automated generation of constraints from use case specifications to support system testing," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 23–33.

[13] C. Wang, F. Pastore, and L. C. Briand, "Automated generation of constraints from use case specifications to support system testing," in *ICST*. IEEE Computer Society, 2018, pp. 23–33.

[14] A. Orso and G. Rothermel, "Software testing: A research travelogue (2000–2014)," in *Proceedings of the on Future of Software Engineering*, ser. FOSE 2014. New York, NY, USA: ACM, 2014, pp. 117–132.

[15] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013.

[16] M. J. E. Cuaresma, J. J. Gutiérrez, M. Mejías, G. Aragón, I. M. Ramos, J. T. Valderrama, and F. J. D. Mayo, "An overview on test generation from functional requirements," *Journal of Systems and Software*, vol. 84, no. 8, pp. 1379–1393, 2011. [Online]. Available: https://doi.org/10.1016/j.jss.2011.03.051

[17] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 31–36.

[18] A. Nayak and D. Samanta, "Synthesis of test scenarios using UML activity diagrams," *Software and System Modeling*, vol. 10, no. 1, pp. 63–89, 2011. [Online]. Available: https://doi.org/10.1007/s10270-009-0133-4

[19] D. Kundu, M. Sarma, and D. Samanta, "A uml model-based approach to detect infeasible paths," *J. Syst. Softw.*, vol. 107, no. C, pp. 71–92, Sep. 2015.

[20] C. Sun, Y. Zhao, L. Pan, X. He, and D. Towey, "A transformation-based approach to testing concurrent programs using UML activity diagrams," *Softw., Pract. Exper.*, vol. 46, no. 4, pp. 551–576, 2016. [Online]. Available: https://doi.org/10.1002/spe.2324

[21] V. Arora, R. Bhatia, and M. Singh, "Synthesizing test scenarios in uml activity diagram using a bio-inspired approach," *Comput. Lang. Syst. Struct.*, vol. 50, no. C, pp. 1–19, Dec. 2017. [Online]. Available: https://doi.org/10.1016/j.cl.2017.05.002

[22] C. Wang, F. Pastore, A. Goknil, L. C. Briand, and M. Z. Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, 2015, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771812

[23] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "NAT2TEST SCR: Test case generation from natural language requirements based on SCR specifications," *Science of Computer Programming*, vol. 95, no. P3, pp. 275–297, Dec. 2014.

[24] T. Yue, S. Ali, and M. Zhang, "Rtcm: A natural language based, automated, and practical test case generation framework," in *Proceedings of International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 397–408.

[25] E. Sarmiento, J. C. Leite, E. Almentero, and G. S. Alzamora, "Test scenario generation from natural language requirements descriptions based on petri-nets," *Electronic Notes in Theoretical Computer Science*, vol. 329, pp. 123 – 148, 2016.