Complete contracts through specification drivers

Alexandr Naumchev*, Bertrand Meyer†
Software Engineering Laboratory
Innopolis University
Innopolis, Russian Federation
†Also Politecnico di Milano
Email: *a.naumchev@innopolis.ru, †Bertrand.Meyer@inf.ethz.ch

Abstract—Existing techniques of Design by Contract do not allow software developers to specify complete contracts in many cases. Incomplete contracts leave room for malicious implementations. This article complements Design by Contract with a simple yet powerful technique that removes the problem without adding syntactical mechanisms. The proposed technique makes it possible not only to derive complete contracts, but also to rigorously check and improve completeness of existing contracts without instrumenting them.

Index Terms—specification driver, abstract data type, Design by Contract, complete contract, Eiffel, Hoare triple, AutoProof

I. INTRODUCTION

The main contribution of this work is a new approach to seamless software development, bridging the heretofore wide gap between two fundamental and widely used techniques: Abstract Data Types (ADTs) and Object-Oriented Programming (OOP). These techniques seem made for each other, but trying to combine them in practice reveals a glaring impedance mismatch. We explain the problem, provide a remedy, and subject it to formal verification.

ADTs [1] are a clear, widely known way to specify systems precisely. OOP [2] is the realization of ADT ideas at the design and programming level, with Design by Contract (semantic properties embedded in the program) providing the connection. At least, that is the accepted view. However, the correspondence is far less simple than this view would suggest. While it would seem natural to use ADTs for specification and OOP for design and implementation, in practice this combination hits an impedance mismatch:

• At the ADT level, some axioms involve two or more commands. For example, an axiom for stacks (the standard example of ADTs, which remains the best for explanatory purposes) will state that if you push an element onto a stack and then pop the stack, you end up with the original stack.

• In a class, the standard unit of OOP, the contracts can only talk about one command, such as push or pop, but not both. Specifically, the postcondition of a command such as push can describe the command’s effect on queries such as top (after you have pushed an element, that element is the new top), but there is no way to refer to the effect on pop as expressed by the ADT axiom.

The present work introduces a practical solution to this mismatch. The essence of the solution is that classes directly reflecting ADTs, such as a class STACK, cannot by themselves capture such multi-command (or "second-degree") ADT axioms, but this does not mean that the OOP approach fails us. The idea will be to introduce auxiliary classes whose role is to "talk about" the features of the basic classes such as STACK (the ones directly corresponding to ADTs). Such a class has features that combine those of basic classes, e.g. a command push_then_pop that works on an arbitrary stack, pushing an element on a stack and then popping the stack. Then the postcondition of push_then_pop can specify that the resulting stack is the same as the original.

We call such features specification drivers by analogy with "test drivers", which are similarly added to the basic units of a system for the sole purpose of testing them. Like test drivers, specification drivers serve purely verification purposes, rather than providing system functionality. The difference is of course that test drivers appear in dynamic verification (testing), whereas specification drivers are for static verification (for example, as in this paper, correctness proofs). But the basic idea is the same.

Specification drivers are not just a specification technique; we also submit them to formal, mechanical verification. As part of the AutoProof formal verification tool [3], we have mechanically proved the correctness of the examples given in this paper.

Section II explains the problem through a working example. Section IV describes the essentials of the solution. Section V compares this approach with other possible ones. Section VI presents our experience with mechanical verification. Section VII draws conclusions and outlines future research prospects.

II. MOTIVATING EXAMPLE

Figure 1 contains the standard ADT specification of stacks. The standard names of the functions are changed in favor of the mechanical verification experiment in Section VI: the existing implementation, to which the experiment is applied, uses exactly these names. Figure 2 contains the result of applying the traditional process of DBC [2] to the specification in Figure 1:

• The name of the class is derived from the name of the ADT it implements.

• The signatures of the implementation features are derivatives of the ADT functions’ descriptions.
Preconditions of the ADT functions go to require clauses of the implementation features.

Postconditions of the implementation features capture ADT axioms A1, A3 and A4.

The create clause lists the implementation feature new to highlight its special mission of instantiating new stacks.

Axiom A2 introduces the problem. The axiom constrains

**Types**

- STACK[G]

**Functions**

- item : STACK[G] → G
- is_empty : STACK[G] → BOOLEAN
- new : STACK[G]

**Axioms**

For any x : G, s : STACK[G]

(A1) item(extend(s,x)) = x
(A2) remove(extend(s,x)) = s
(A3) is_empty(new)
(A4) not is_empty(extend(s,x))

**Preconditions**

(P1) remove(s : STACK[G]) require not is_empty(s)
(P2) item(s : STACK[G]) require not is_empty(s)

Fig. 1. ADT specification of stacks

```java
class STACK_IMPLEMENTATION [G] {
create new -- Marking new as a creation feature
feature
extend (x: G) -- Extending with a new element
  do
    a1: item = x
    a4: not is_empty
  end
remove -- Removing the topmost element
  require
    p1: not is_empty
  do
    end
item: G -- The topmost element
  require
    p2: not is_empty
  do
    end
is_empty: BOOLEAN -- Is the stack empty?
new -- Instantiating a stack
  do
    a3: is_empty
  end
}
```

Fig. 2. Applying the traditional process of DbC to the stacks ADT specification

class STACK_IMPLEMENTATION [G]
create new
feature
extend (x: G) -- Extending with a new element
  do
    item := x
    is_empty := False
  ensure
    a1: item = x
    a4: not is_empty
  end
remove -- Removing the topmost element
  do
    is_empty := True
  end
item: G -- The topmost element
  is_empty: BOOLEAN -- Is the stack empty?
new -- Instantiating a stack
  do
    is_empty := True
  ensure
    a3: is_empty
  end
end

Fig. 3. Underspecified postconditions may lead to invalid implementations

two functions simultaneously, extend and remove: the former one should do nothing but extend the stack with the given element, and the latter should do nothing but remove the topmost element of the stack. As a consequence, it is not possible to capture the axiom in a single implementation feature postcondition. Postconditions operate on two objects: the target object before calling the feature and the target object after invoking the feature. If the feature has formal parameters, they also parameterize the postcondition. Axiom A2 involves three stacks: the original one s, s1 resulting from applying function extend to s, and finally s2 resulting from applying remove to s1. Formally:

\[ \forall s, s_1, s_2 : STACK[G]; x : G \bullet (s_1 = extend(s, x) \land s_2 = remove(s_1) \Rightarrow s_2 = s) \]

Or, writing the quantified expression in terms of postconditions:

\[ (Post_{extend}(s, s_1, x) \land Post_{remove}(s_1, s_2)) \Rightarrow s_2 = s \quad (1) \]

On one hand, it is not possible to capture A2 in a single postcondition. On the other hand, postconditions of extend and remove should exist and be strong enough to satisfy Equation 1.

Failures to capture such important properties as A2 in postconditions leave room for invalid implementations. In particular, inability to capture axiom A2 makes it possible to implement stacks which store only the last added element and thus are useless as data containers. Still, such an implementation satisfies all the other axioms as its postconditions capture them.
Figure 3 depicts such an invalid implementation. For the sake of simplicity, it ignores preconditions, but this does not render the reasoning invalid: an empty precondition defaults to true, the weakest conceivable precondition. According to the rule of consequence for preconditions [4], correctness against a weaker precondition implies correctness against a stronger one. Submitting the class STACK_IMPLEMENTATION to AutoProof confirms the point: the tool successfully proves "correctness" of the implementation.

For purist developers the problem of underspecified postconditions may easily become a reason for not using them at all. Intuitively, it seems better to keep all the properties written in a single place, and the described problem prevents doing this: although it is possible to capture some ADT axioms in postconditions, some of them will have to exist in separate documents and thus carry the risk of misuse and all the associated traceability costs.

The example in Figure 2 translates axiom A1 directly to the postcondition of the implementation feature extend. Is it in fact the only way to do the translation of the axiom? A closer look at the original axiom and its translation in Figure 2 reveals two facts:

- The axiom uses the function extend in a sense of applying it, while its translation in Figure 2 specifies the implementation feature directly without invoking it.
- The axiom uses an explicit stack instance s, while the translation implicitly operates on the current object described by class STACK_IMPLEMENTATION[G].

Is it possible to devise a translation of axiom A1 that would be closer to the origin?

Existing techniques of DbC completely ignore a large family of program constructs: features with pre- and postconditions whose only purpose is to serve as proof obligations. Such features do not implement any ADT functions and are not to be invoked. Instead, they are intended solely for static verification. Figure 4 gives an example. The feature extend_updates_item is an alternative translation of axiom A1. It possesses the following properties:

- It operates on explicit objects s and x.
- It uses an explicit invocation of implementation feature extend.

The example in Figure 4 takes the whole feature extend_updates_item as the translation of the axiom, as opposed to the one in Figure 2 where the axiom is captured with the assertion item = x in the postcondition of implementation feature extend.

Using this approach, it is possible to capture axiom A2 in the form of the feature remove_then_extend in Figure 5. Again, the whole feature is the translation of the axiom. The feature is_equal defines an equivalence relation over run time objects representing stacks. It is declared by default in all Eiffel classes and compares its operands by value. The notion of equality deserves a separate analysis; Section IV-B gives the details.

Henceforth, this article will use the term specification drivers for specified features serving as translations of certain ADT axioms. A specification driver may be proven correct only if the implementation features it invokes have strong enough postconditions. Consequently, specification drivers, as their name suggests, drive specifying stronger postconditions.

The present section derives the complete set of specification drivers for the stacks ADT (Figure 1). This set includes not only specification drivers that directly represent the original axioms of stacks because some specification drivers stem from a fundamental difference between ADT specifications and object-oriented programs: in the former it is not possible to have more than one occurrence of one and the same abstract stack, while in the latter it is possible to instantiate two run time objects denoting one and the same abstract stack. Section IV-B and Section IV-C discuss the issue in detail and derive additional specification drivers caused by it.

A. ADT axioms

Specification drivers do not bring any functional value to the system: they exist only to be eventually discharged as proof obligations. Consequently, they should not pollute implementation classes like STACK_IMPLEMENTATION in Figure 2. Concerning where to store them, the simplest option is to create a separate class within the source code project. The ADT AXIOMS SPECIFICATION DRIVERS class in Figure 6 contains specification drivers capturing the ADT axioms of stacks. This class is generic: since it talks about instances of a generic concept, STACK IMPLEMENTATION [G] in this case, it needs to assume existence of type G to keep the genericity. The [NONE] clause suggests that the features listed within the corresponding feature block do not supply any useful functionality. The deferred keyword in front of the class declaration suggests that it is not possible to instantiate any objects of this class, which
makes sense as the class serves as a document containing specification drivers rather than a blueprint for creating run time objects.

B. Equivalence

It is possible to see that the specification drivers in Figure 6 use two different operators for objects comparison: = and is_equal, while the original ADT specification in Figure 1 invokes only =. This section discusses the difference between comparing instances of ADTs and comparing objects instantiated from object-oriented classes and introduces a set of specification drivers capturing the difference.

ADT specifications operate on sets of instances in the mathematical sense of the word "set": an abstract data type cannot contain two instances of one and the same abstract object. For example, the range of the function new consists of the only stack instance, which is the empty stack, as axiom A4 suggests. When an object-oriented program is running, it is perfectly fine for it to have two run time objects in its memory denoting one and the same instance of the ADT.

Consequently, run time objects form not a set of abstract objects, but a multiset, or bag [5]. That is why there are two different comparison operators: the = operator checks whether the operands refer to identical run time objects, and is_equal checks whether the objects referenced by the operands represent the same instance of the ADT implemented by the class. As a consequence, if specification drivers representing ADT axioms use the feature is_equal, the corresponding implementation class should redefine the feature and its postcondition should be strong enough to satisfy the definition of equivalence relations. A relation over stacks is an equivalence relation if and only if it possesses the following properties:

- Reflexivity: every stack is equal to itself.
- Symmetry: if stack s1 is equal to stack s2, then s2 is equal to s1 as well.
- Transitivity: if stack s1 is equal to stack s2, and s2 is equal to s3, then s1 is equal to s3.

Fig. 7. Creating two instances of the empty stack

| deferred class ADT_AXIOMS_SPECIFICATION_DRIVERS [G] |
| feature {NONE} |
| axiom.a1 (s: STACK_IMPLEMENTATION [G]; x: G) |
| do |
| s.extend (x) |
| ensure |
| s.item = x |
| end |
| axiom.a2 (s1, s2: STACK_IMPLEMENTATION [G]; x: G) |
| require |
| sl.is_equal (s2) |
| do |
| s1.extend (x) |
| sl.remove |
| ensure |
| sl.is_equal (s2) |
| end |
| axiom.a3 (s: STACK_IMPLEMENTATION [G]; x: G) |
| do |
| s.extend (x) |
| ensure |
| not sl.is_empty |
| end |

Fig. 6. Specification drivers capturing the axioms of stacks

As Figure 8 illustrates, the three properties may be captured by a separate class created specifically for this goal. If all the features of class EQUIVALENCE_SPECIFICATION_DRIVERS are correct, then the postcondition of is_equal indeed defines an equivalence relation over run time objects instantiated from STACK_IMPLEMENTATION [G].

It is worth noting that because equivalence definition is static, specification drivers for equivalence may be generated automatically for every class.

C. Well-definedness

The ADT specification in Figure 1 lists certain functions over stacks. It is necessary to ensure that they remain functions in the presence of an equivalence relation. Invoking a given
Deferred class WELL_DEFINEDNESS_SPECIFICATION_DRIVERS [G]
feature {NONE}
new.is.well_defined (s1, s2: STACK_IMPLEMENTATION [G])
  require
    s1.is.empty
    s2.is.empty
    s1 ≠ s2
  do
    s1.is.equal (s2)
  ensure
  s1.is_empty = s2.is_empty
  end
is_empty.is.well_defined (s1, s2: STACK_IMPLEMENTATION [G])
  require
    s1.is.equal (s2)
    s1 ≠ s2
  do
    s1.is_empty = s2.is_empty
  ensure
  s1.is_empty = s2.is_empty
  end
item_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
  require
    not s1.is_empty
    not s2.is_empty
    s1.is.equal (s2)
    s1 ≠ s2
  do
    s1.item = s2.item
  ensure
  s1.is.item_equal = s2.is.item
  end
extend_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G]; x: G)
  require
    s1.is.equal (s2)
    s1 ≠ s2
  do
    s1.extend (x)
    s2.extend (x)
  ensure
    s1.is_equal (s2)
  end
remove_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
  require
    not s1.is_empty
    not s2.is_empty
    s1.is_equal (s2)
    s1 ≠ s2
  do
    s1.remove
    s2.remove
  ensure
    s1.is_equal (s2)
  end

Fig. 9. Specification drivers for well-definedness

D. Complete contracts

Although some works ([7], [8]) talk about contract (in)completeness, they do not define this notion precisely. In light of the fundamental difference between ADT specifications and object-oriented programs, which causes the notion of equivalence over run time objects to appear ([Section IV-B]), the definition cannot be implicitly equal to the definition of sufficiently complete ADT specifications [9] and needs to be written down explicitly.

As the other details of the original definition in [2] do not bring any value to the discussion, this article uses a simplified definition of a contract.

Definition 4.1: A contract is a set composed of all pairs of the form \((\text{Precondition}(f), \text{Postcondition}(f))\) for every implementation feature \(f\).

This definition ignores the possible presence of class invariants as it is always possible to get rid of them by appending to pre- and postconditions of the implementation features.

Definition 4.2: A contract is correct if and only if:

- Its postconditions are strong enough to ensure correctness of the specification drivers derived from the input ADT axioms ([Section IV-A]).
- In the event that specification drivers for the input ADT axioms use equivalence, its postconditions are strong enough to ensure correctness of the specification drivers for equivalence ([Section IV-B]).

Definition 4.3: A contract is well-defined if and only if its postconditions are strong enough to ensure correctness of the specification drivers for well-definedness ([Section IV-C]).

Definition 4.4: A contract is complete if and only if it is correct and well-defined.
V. RELATED WORK

Doctoral thesis [7] uses features with pre- and postconditions for checking completeness of model-based contracts (discussed later in this section). The definition of a complete model-based contract is not related to the definition of completeness in Section IV-D. According to [7], completeness is what the current article calls well-definedness, expressed in terms of abstract mathematical concepts.

Although the specification driver approach allows capturing ADT axioms in their original form, it does specify how to actually build complete contracts having a set of specification drivers. As Section II suggests, in many cases it is not possible to specify strong enough postconditions in terms of the ADT specification itself. This is where the need for representation appears: the implementation class has to stick to some already implemented data structure in order to enable stronger postconditions expressed in terms of this data structure. The problem of choosing an ideal representation has been aptly handled in multiple publications, therefore the present article does not propose its own methodology, but chooses instead to reference these publications.

Work [10] shows that it makes sense to use mathematical abstractions for representations: for example, it seems reasonable to think about stacks as mathematical sequences. That work also shows how to prove correctness against contracts strengthened with precise mathematical abstractions. Work [8] introduces the Mathematical Model Library (MML) - Eiffel library containing core abstractions: sets, sequences, bags, tuples etc. A more recent work [11] introduces EiffelBase2, a usable library of essential data structures, including stacks, represented as mathematical abstractions from MML. EiffelBase2 is fully verified with the AutoProof verifier. The underlying verification methodology [12] assumes writing quite a number of assertions related to program execution semantics, so giving complete examples here would introduce confusion rather than clarity. Instead, Figure 10 presents the idea in a nutshell. The STACK.SEQUENCE.IMPLEMENTATION class is the abstract model of stacks from the EiffelBase2 standpoint. EiffelBase2 equips classes implementing stacks with the sequence attribute and strengthens postconditions of the implementation features in terms of it. Class MML.SEQUENCE cannot be instantiated into any run time objects and exists only for verification purposes: it maps directly to the data structure representing mathematical sequences in the underlying proving engine. The sequence attribute is further connected to meaningful data structures by means of abstraction and refinement techniques [13]. Works [11] and [7] give more implementation details.

In Figure 10 the implementation features are formally defined with assertions over the sequence attribute (marked with the "definition" tag) added to the features’ postconditions. The comparison feature is_equal is redefined so that two stacks are considered equal if and only if the sequences representing them are equal. Two sequences are considered equal if and only if their sizes are equal and they contain same objects. The feature extended models a sequence where an object is appended to the target sequence on to which the feature is invoked; feature but_last models the target sequence, but without the last element; feature last models the element added to the target sequence last; feature is_empty models the indication whether the target sequence is empty or not; finally, feature count models the size of the target sequence.

Mathematical concepts from MML are abstract, but they still form particular representations in EiffelBase2, though mathematically precise. The concept of model-based contracts helps to specify complete contracts, but does not say how to rigorously check contracts with representations for complete-
ness. Furthermore, it fails to define what complete contracts are. The notion of specification drivers bridges this gap. All the specification drivers derived in the present article are expressed in terms of the original ADT specification (Section IV-A) plus the abstract equivalence (Section IV-B and Section IV-C), whose presence is inevitable due to the nature of computing which allows programs to keep in their memory several instances of one and the same abstract object. They do not require making any assumptions about possible representations and enable defining complete contracts precisely.

VI. PROVING CONTRACTS COMPLETENESS

It is possible to give a manual proof of completeness of the contract depicted in Figure 10. Fortunately, this work may be done automatically. This advantage makes it possible to apply the specification drivers approach to legacy implementations. Indeed: if there is a source code project with a number of classes in it, then it is possible to devise an additional class, write all the applicable specification drivers into it and submit the resulting class to the prover. Instead of showing how to derive complete contracts having a set of specification drivers from scratch, the article shows how to apply the approach to existing contracts.

The EiffelBase2 library seems to be a natural choice for the experiment. The library contains a complete implementation of stacks specified as mathematical sequences. The corresponding implementation class is V.LINKED_STACK. In order to perform the experiment, it is necessary to take the stacks specification drivers from Section IV and modify them so that the name of the implementation class would be V.LINKED_STACK instead of STACKS_IMPLEMENTATION. The specification driver axiom_a4 comes with a pitfall: the V.LINKED_STACK class does not introduce its own creation feature, but redefines the default creation feature defined for all classes. Hence, the create Result.new instruction is not applicable here; one should use create Result instead. After these modifications, the specification drivers should successfully compile and be ready for verification.

The initial verification attempt using AutoProof will result in numerous precondition violations. As Section V suggests, the verification methodology [12] behind AutoProof assumes writing additional non-stack related assertions. For example, the extend is_well_defined specification driver can be verified by AutoProof only in the form depicted in Figure 11. The five assertions in the beginning of the require precondition clause seem to be worth explaining them briefly. The s1.is_wrapped assertion says that reference s1 is assumed to be non-void and not participating in any call; the s1.observers.is_empty assertion says that the set of objects interested in the state of s1 should be empty - it is a part of the precondition of feature extend of class V.LINKED_STACK; finally, the modify([s1, s2]) assertion is a frame specification: it says that the enclosing feature, extend is_well_defined in this case, is going to modify objects referenced by s1 and s2 (square brackets [] denote set constants in Eiffel). The precondition needs the modify assertion because the extend is_well_defined feature uses feature invocations with side effects, extend in this case, on references s1 and s2.

Although the verification failures caused by the absence of these assertions do not bear any relation to stacks, they uncover certain weaknesses in the verification methodology: namely, the defaults do not seem sufficiently reasonable. For example, a violation of the s1.is_wrapped assertion would detect a callback situation, and callbacks are not so common as to assume them by default. The observers.is_empty requirement makes extending stack objects applicable only in situations when no other objects depend on their states. The modify frame specification may be generated automatically based on the presence of invocations with side effects in the implementation body.

After complementing the specification drivers with all necessary assertions related to verification methodology and rerunning AutoProof, it uncovers some stack-related issues. This is visible from the fact that this time the verification errors come from the postconditions. Namely, AutoProof fails to prove correctness of all the verification drivers from classes EQUIVALENCE_SPECIFICATION_DRIVERS and WELL_DEFINEDNESS_SPECIFICATION_DRIVERS as well as verification driver axiom_a2 from the ADT_AXIOMS_SPECIFICATION_DRIVERS class. As all of these specification drivers involve implementation feature is_equal, the first guess is that V.LINKED_STACK does not redefine it. This guess appears to be right: the class defines its own custom feature for comparing run time objects, but does not redefine the standard comparison feature in terms of the new one. Giving this flaw’s fix here would not bring much value to the discussion, so it seems better to move on. After redefining feature is_equal, AutoProof succeeds in proving classes ADT_AXIOMS_SPECIFICATION_DRIVERS and EQUIVALENCE_SPECIFICATION_DRIVERS completely, but still fails to prove specification driver new.is_well_defined from the WELL_DEFINEDNESS_SPECIFICATION_DRIVERS class. As this specification driver uses the is_empty implementation feature, it falls under suspicion. Apparently, its postcondition does not have a clause corresponding to the definition clause in its abstract model in Figure 10. After fixing this flaw, everything verifies successfully, including the V.LINKED_STACK implementation class.
VII. Conclusions and Further Work

The article makes the following main contributions:

- Presents the specification driver approach for encoding ADT axioms, which are not possible to encode using traditional DbC techniques.
- Illustrates the process of axiomatizing abstract equivalence using the new approach.
- Introduces an exhaustive definition of contract completeness.
- Demonstrates how to apply completeness checks to legacy implementations.

The new approach allows adding, changing or removing ADT axioms at any given moment of the development process without necessarily modifying the implementation classes. Although specification drivers occupy separate classes completely disjoint from implementation classes, they are simultaneously expressed in terms of objects instantiated from the implementation classes. The result is a seamless integration of software axiomatization and implementation driven by automatic verification of functional correctness. Attempts to check specification drivers can uncover weak postconditions of implementation features. Once strengthened, these postconditions potentially yield firmer executable instructions.

In light of the presence of different kinds of specification drivers described in Section IV it seems feasible to propose the following changes to the Eiffel Verification Environment tool set:

- Develop a template for fast creation of classes intended to keep specification drivers.
- Automate generation of specification drivers for equivalence and well-definedness.
- Revise verification methodology underlying AutoProof: in essence, specification drivers are a new syntactical specification construct, which may potentially remove some particularly egregious verification challenges.

Work [14] introduces the notion of multirequirements, and work [15] illustrates how to apply this notion in practice. The underlying idea is that a separate item in a software requirements document should be expressed using several interwoven notations, e.g. natural language, graphical form and formal notation. For the formal notation, it was suggested to use a rather expressive programming language. The present paper talks about expressing ADT axioms in a programming language with pre- and postconditions. Since ADT specifications are one of the languages for expressing software requirements, it makes sense to revisit the original multirequirements approach to see how the idea of specification drivers could improve it.

The idea of specification drivers was inspired mostly by seminal works [13] and [2], and driven by the will to unify requirements and code seeded in the work [14].

Acknowledgment

The authors would like to thank Innopolis University for supporting the Software Engineering Laboratory where the research resulted in this work is taking place.

References


Special thanks goes to Daniel Johnston from MSIT-SE Program at Innopolis University, who kindly agreed to proofread this paper line-by-line.