Seamless Development Applicability: an Experiment

Alexandr Naumchev <a.naumchev@innopolis.ru>,
Innopolis University, Innopolis, Russian Federation

Abstract. Requirements and code, in conventional software engineering wisdom, belong to entirely different worlds. The usual view in software engineering considers requirements documents and source code as different artifacts, under the responsibility of different people. This approach, however, introduces communication overhead, and raises the question of how to keep the various artifacts consistent when either of them needs to change. A change introduced to any of the mentioned artifacts needs to be synchronized with the others. At some point the control is inevitably lost: for example, a critical bug is found during the software operation, and the software developers dig into the fixing process directly, because there is no time to wait until the requirements analysts and system architects update their documents to let the developers actually fix the problem. Is it possible to unify the two worlds? A unified framework could help make software easier to change and reuse. To explore the feasibility of such an approach, the case study reported here takes a classic example from the requirements engineering literature and describes it using a programming language framework to express both domain and machine properties. The paper describes the solution, discusses its benefits and limitations, and assesses its scalability.

Keywords: software engineering; requirements specifications; multirequirements; Eiffel

DOI: 10.15514/ISPRAS-2015-27(3)-4

For citation: Naumchev Alexandr. Seamless Development Applicability: an Experiment. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 3, 2015, pp. 57-72. DOI: 10.15514/ISPRAS-2015-27(3)-4.

1. Introduction

Nowadays the dominating view on the software engineering discipline includes an implicit assumption that engineering the requirements, designing the architecture and implementing the code are all separate activities. "Separate" means that an engineer performs only one of them at the same time and produces different artifacts as the output. This implicit assumption is cultivated by the top software engineering schools who promote the idea explicitly enough to push it to the students' subconscious level.

1.2 Problems with the Current Approach

The usual view in software engineering considers requirements documents and source code as different artifacts, under the responsibility of different people. This approach, however, introduces communication overhead, and raises the question of how to keep the various artifacts consistent when either of them needs to change. A change introduced to any of the mentioned artifacts needs to be synchronized with the others. At some point the control is inevitably lost: for example, a critical bug is found during the software operation, and the software developers dig into the fixing process directly, because there is no time to wait until the requirements analysts and system architects update their documents to let the developers actually fix the problem. The problem is partially solved with complicated configuration management, which is expensive and difficult to maintain, and may serve as a source of evil as well: there are so called ”technical commits”. Only senior developers are allowed to make them, and the basic idea is that such commits do not have to be linked to some task, bug or user story (if the team practices Agile). Quite often the technical commits contain basically whole new features or big chunks of code not linked to any document.

Why should we try to minimize gaps between requirements and code? At the very least because successful software evolves. The customers want more features, they want to improve existing features, and they want to know how much money it will cost and how much time it will take. If it is possible to relate the ideas to the artifacts, then by comparing complexity of some new idea with an existing one, already implemented, it will be possible to estimate the resources required for implementing the new idea.

The list of the problems discussed above does not pretend to be exhaustive of course, but it should be sufficient to start thinking about changing the overall approach.

1.2 Existing Solutions

Typically the problems from Section 1.1 are resolved by carefully choosing appropriate notations for every development life cycle phase. The selection criteria include possibility of establishing traceability links between different notations. Each phase requires the output of the previous phase on its input and on its output produces the input for the next phase. In [2], authors give an example of applying this approach. This work also contains an overview of the most popular notations used in formal software development. For instance, the software development case described in work [2] uses natural language for requirements document, RSML [7] for specification document, Event-B [1] for developing software formal model, formalizing the requirements and formally verifying the model against the requirements. Finally, EventB2Java [8] generates executable Java source code equipped with JML specs from a model expressed in Event-B. For moving from the requirements document to the specification document the Problem Frames

Approach [5] is applied. The latter method produces a problem frames model on the output.

Needless to say, such approach requires people with very rich set of skills: for example, to produce a specification document expressed in RSML, the responsible person also has to understand the Problem Frames Approach. In a similar fashion the person responsible for modeling in Event-B also has to be proficient with RSML, and so on.

As a software engineer we should not forget why there is a huge gap between requirements and code at all. The fundamental reason is in limited expressive power of programming languages compared to expressive power of any natural language. That is why there are many ”intermediate” notations serving for smooth transition from natural language requirements to source code; that is why the coding phase and the requirements engineering phase typically have tiny overlaps in time, and there are other software development life cycle phases between them. If it was possible to express any executable requirement using a subset of some programming language, then the problem would disappear.

1.3 Unified View on Software: The Hypothesis

It is possible to design such a software development process that:

1. By specifying the requirements the analyst at the same time will also design the solution
2. The resulting document may be linked in an intuitive way to an algorithmic implementation
3. The resulting implementation will be formally provable against the requirements specification
4. Small change in the requirements specification will cause proportionally small change in the design and the implementation

Parts 1, 2 and 3 promote consistency between the requirements, design and implementation; part 4 promotes predictability of resources estimations.

1.4 How to Test the Hypothesis

The following process seems to be feasible for testing adequacy of the stated hypothesis:

1. Propose a candidate process
2. Select some real projects which are presumably prone to the problems stated in section 1.1
3. Apply the proposed process to the selected projects and see how it goes

In [11] Meyer sketched such a process based on using object orientation for representing the relationships between the conceptual objects mentioned in the requirements document. The basic idea was to have an object-oriented code along with the natural language description of a requirements item. Each code fragment in its turn may be represented graphically as a BON diagram [15].

The main problem with [11] was the example used for the demonstration purposes: it was self-referential. That is, it contains “requirements for the requirements”.

Nevertheless, it demonstrates that object orientation contributes to understanding the relationships between the objects. However, requirements (in their general form) are beyond this: to specify requirements, as described by Jackson and Zave in [6], is also to specify all allowed sequences of events associated with a given problem area.

The present work provides an example of how one could combine approaches from [11] and [6] by adding fully-fledged contracts, both in their classical and model-based semantics, to the requirements specification notation. More precisely, it contains every requirements item from the Zoo Turnstile example discussed in [6] represented using the model-based [13] contracts-equipped [10] object-oriented [9] notation (Eiffel).

2. Theoretical and Technical Background

2.1 Design By Contract

A comprehensive description of Design By Contract is given in [10]. Design By Contract integrates Hoare-style assertions [3] within object-oriented programs [9]. This concept assumes that each class feature (member), is equipped with its pre- and postcondition, which are predicates on the class. The postcondition has to hold whenever the precondition held and the feature finished its computation before the next feature is invoked. The class itself is equipped with an invariant expression which holds in all states of the corresponding instantiated objects.

2.2 Model-Based Contracts

If classical contracts are for constraining the data actually held by run-time objects, model-based contracts are ”meta” contracts for constraining the objects as mathematical entities (sets, sequences, bags, relations etc.), and the corresponding mathematical representations are not actually instantiated at run-time as parts of the objects. Model-Based Contracts are needed when it is not possible to capture all the nuances by means of classical contracts. Some examples of such situations and a comprehensive description of the concept is given in the PhD thesis [13].

2.3 AutoProof

Object-oriented classes constrained with contracts (both classical and model-based) may be formally verified using an automation called AutoProof [14]. AutoProof traverses over the class features and proves formally that the precondition conjuncted with the class invariant ensures the postcondition together with the class invariant after the feature application. If all the class features are verified, then the class is considered verified.

3. Unifying the Two Worlds: an Example

This section shows the approach at work. It takes the example introduced by Jackson and Zave in [6] in 1995 and specifies the example using Eiffel programming language [16] as a formal notation. Originally this example was used to demonstrate the process of deriving specifications from requirements, and the unified approach captures all the nuances of this process.

3.1 Example Overview

The authors of [6] start with giving the overall context: *"...Our small example concerns the control of a turnstile at the entry to a zoo. The turnstile consists of a rotating barrier and a coin slot, and is fitted with an electrical interface..."* This small paragraph describes mostly relationships between the conceptual objects and thus may be expressed in the style of work [11]:

```
deferred class ZOO
feature
    turnstile : TURNSTILE
end

deferred class TURNSTILE
feature
    coinslot : COINSLOT
    barrier : BARRIER
invariant
    coinslot.turnstile = Current
    barrier.turnstile = Current
end

deferred class COINSLOT
feature
```

```
   turnstile : TURNSTILE
invariant
   turnstile.coinslot = Current
end

deferred class BARRIER
feature
   turnstile : TURNSTILE
invariant
   turnstile.barrier = Current
end
```

Fig. 1: Expressing the context formally

Translating this code (fig. 1) back to English using the object-oriented semantics results in almost the same initial description: ”A ZOO has a TURNSTILE turnstile; a TURNSTILE has a COINSLOT coinslot and a BARRIER barrier so that coinslot has Current TURNSTILE as turnstile and barrier has Current TURNSTILE as turnstile...” COINSLOT and BARRIER hold references to the TURNSTILE instances in order to capture the *”electrical interface”* phenomena: the word ”interface” means something over which the parties are able to communicate with each other; communicating means sending messages to each other, and to send message to someone in the object-oriented world is to take the corresponding instance and perform a qualified call. So at the very least the parties should hold references to each other to be able to communicate in two directions.

3.2 The Designation Set

After stating the problem context the authors of [6] describe a *designation set*. Each designation basically corresponds to a separate type of events observed in the problem area. The designations are provided in form of the predicates:

- **Push**(*e*): In event *e* a visitor pushes the **barrier** to its intermediate position
- **Enter**(*e*): In event *e* a visitor pushes the barrier fully home and so gainsentry to the **zoo**
- **Coin**(*e*): In event *e* a valid coin is inserted into the **coin slot**
- **Lock**(*e*): In event *e* the **turnstile** receives a locking signal
- **Unlock**(*e*): In event *e* the **turnstile** receives an unlocking signal

The representation of this designation set provided below (fig. 2) uses Eiffel features names as labels for the events types (entities introduced earlier are not repeated afterwards). The aforementioned natural language descriptions provide heuristics on which feature should be added to which class (the association is highlighted with **bold**). Not only different types of events, but also the history of the corresponding events, are designed using Eiffel features. For example, *enters* : *MML_SEQUENCE* [*INTEGER*_64] is a sequence of moments in time expressed in milliseconds when events of type *enter* took place. *model* annotation says that *enters* feature will be used for expressing the model-based part of the contract (model-based contracts were introduced in section 2.2). *MML_SEQUENCE* is a class from the *MML* (Mathematical Modeling Library) and denotes mathematical sequence. *MML* was designed specially to express model-based contracts. Although it is possible to instantiate some simple objects from these classes (like a sequence containing one element), one cannot modify the instances.

```
 note
   model: enters deferred class ZOO
feature
   enter deferred ensure
      enters.but_last ~ old enters
      enters.last > old enters.last
   end
   enters : MML_SEQUENCE [INTEGER_64 ]
end
 note
   model: locks, unlocks deferred class
TURNSTILE feature
   lock deferred ensure
      locks.but_last ~ old locks
      locks.last > old locks.last
   end
   unlock
   deferred
   ensure
      unlocks.butlast ~ old unlocks
      unlocks.last > old unlocks.last
   end
   locks : MML_SEQUENCE [INTEGER_64]
   unlocks : MML_SEQUENCE [INTEGER_64]
 end
```

```
note
  model: coins
deferred class COINSLOT
feature coin deferred ensure
    coins.butlast ~ old coins
    coins.last > old coins.last
  end
  coins: MML_SEQUENCE[INTEGER_64]
end
note
  model: pushes deferred class BARRIER
feature
  push deferred ensure
    pushes.butlast ~ old pushes pushes.last >
    old pushes.last
  end
  pushes: MML_SEQUENCE[INTEGER_64]
end
```

Fig. 2: Specifying the designation set formally

The *deferred* keyword is used to highlight that the events are only specified formally, without specifying the corresponding operational reactions of the software to the events. The *ensure* clause is used to specify what conditions should be satisfied after reacting on an event. These specifications are intuitively plausible: the events history should be complemented with the new event occurrence, and the time of the new event should be strictly bigger than the time of the previous event.

3.3 Shared Phenomena

The authors of [6] introduce the notion of shared phenomena that is, the phenomena visible to both the world and the machine (the notions of the world and the machine were introduced by Jackson in [4]). In the present approach this notion is covered by using the “has a” relationships between the *ZOO* and the *TURNSTILE* classes, accompanied with the model-based contracts. Namely,since a *ZOO* has a turnstile as its feature, it can see any phenomena hosted by the turnstile: *locks, unlocks, coins, pushes*. And since a *TURNSTILE* does not hold any references to a *ZOO*, it can not observe nor control the *enter* events modeled by *ZOO*.

3.4 Specifying the System

All the properties of the problem derived in [6] be they optative or indicative descriptions can be conceptually divided into the three main categories.

Properties which hold at any moment in time An example of such properties is the *OPT1* requirement saying that entries should never exceed payments (the authors of [6] use *OPT∗* for labeling properties expressed in an optative mood). Within the present approach this requirement can be expressed in the following way (fig. 3):

```
deferred class ZOO
feature
   enters : MML_SEQUENCE [INTEGER_64]
   turnstile : TURNSTILE
invariant
   enters.count <= turnstile.coinslot.coins.count
end
```

Fig. 3: Entries should never exceed payments

The ”something always holds” semantics fits perfectly into the semantics of Eiffel invariant: ”something holds in all states of the object”.

Properties which hold depending on the type of the next event to occur The indicative property *IND*2 saying that it is impossible to push the barrier if the turnstile is locked will serve as an example. Below (fig. 4) is the corresponding specification:

```
deferred class BARRIER
feature push require
      not turnstile.unlocks.is_empty
      not turnstile.locks.is_empty implies
         turnstile.unlocks.last > turnstile.locks.last
   deferred end
   pushes : MML_SEQUENCE [INTEGER_64]
end
```

Fig. 4: It is impossible to use locked turnstile

The initial description is divided into the two different claims: first, the turnstile should be unlocked at least once, and second, if the turnstile has ever been locked, the last unlock should have occurred later than the last lock.

Real Time Properties The authors of [6] derive several timing constraints on the events. For example, the *OPT*7 requirement says that the amount of time between the moment when the number of the barrier pushes becomes equal to the number of coins inserted and the moment when the turnstile is locked should be less than 760 milliseconds. It is possible to make this property finer grained. First (fig. 5), if after the next *push* event the number of pushes becomes equal to the number of coins, then after reacting on the *push* event the turnstile should be locked at some point before the next *push* event occurs.

```
deferred class BARRIER
feature
   turnstile : TURNSTILE push
   deferred ensure
      (old turnstile.unlocks.last > old turnstile.locks.last
      and
      pushes.count = turnstile.coinslot.coins.count)
            implies turnstile.locks.last > pushes.last
   end
   pushes : MML_SEQUENCE [INTEGER_64]
end
```

Fig. 5: The machine locks the turnstile timely

Second (fig. 5), if the last *lock* event occurred later than the last *push* event, then thetime distance between them is smaller than 760.

```
deferred class TURNSTILE
feature
   barrier : BARRIER
   locks : MML_SEQUENCE [INTEGER_64] unlocks : MML_SEQUENCE [
   INTEGER_64]
invariant
   locks.last > barrier.pushes.last implies
      (locks.last - barrier.pushes.last) < 760
end
```

Fig. 5: The machine locks the turnstile timely

3.5 Specifying the "Unspecifiable"

One of the requirements mentioned in [6] was *OPT*2 saying that the visitors who pay are not prevented from entering the Zoo. The authors give only informal statement of this requirement:

$$\forall v,m,n \bullet ((Enter\#(v,m) \wedge Coin\#(v,n) \wedge (m < n)) \Rightarrow \text{'The machine will not prevent another Enter event'}$$

The antecedent of this implication should be read like "number of entries is less than the number of coins inserted". In the present specification system thisrequirement can be formalized easily (fig. 6).

```
deferred class ZOO
feature
   enter
   require
      enters.count < turnstile.coinslot.coins.count
   deferred end
   enters : MML_SEQUENCE[INTEGER_64]
end
```

Fig. 6: The turnstile let people who pay enter

It works because semantically the *require* clause specified above is the strongest precondition of the *enter* feature. That is, if some class inherits from *ZOO* and redefines the *enter* feature, it will be allowed to redefine the precondition by using only the *require else* clause that weakens the precondition by ”or”-ing it with the original one. And so, if the *enters.count* < *turnstile.coinslot.coins.count* condition is satisfied, the precondition of the *enter* feature will always be satisfied,thus allowing an *enter* event to occur.

Not only this specification formalizes *OPT* 2 it also ensures satisfaction of *OPT* 1 (together with the *ensure* clause for the *enter* feature introduced earlier): indeed, if the number of enters is always strictly smaller than the number of coins inserted before any *enter* event occurrence, then after the *event* occurrence the number of entries will not exceed the number of coins inserted.

In the process of research the author of the present work identified that the aforementioned reasoning about formalizing *OPT* 2 requirement is farfetched and is not scalable. For example, if Zoo management decides to install one more appliance for controlling Zoo entrance, and the corresponding requirements will enrich the precondition of the *enter* feature, the whole reasoning will be invalidated. The author found more scalable and intuitively plausible way to formalize this requirement in Eiffel. The corresponding formalism will be available in work [12].

4. Conclusion

The specification method discussed in this work is suitable not only for formalizing statements which were also formalized in [6], but also for formalizing statements which cannot be formalized with the classical tools used in [6]. Not only the requirements specification items were expressed, but also the object-oriented blueprint was built ready to equip it with code actually doing something useful. Such implementation exists and is available here: https://github.com/anaumche/Zoo-Turnstile-Multirequirements.

4.1 Pros & Cons

It is necessary to evaluate the method against the characteristics of the hypothesis introduced in section 1.3:

1. Simultaneity of specifying the requirements and building the design: indeed, all the code fragments corresponding to different specification items merged together will bring a complete design solution available at https://github.com/anaumche/Zoo-Turnstile-Multirequirements (the classes ending with " abstract").
2. Traceability between the specification and the implementation: the classesending with " concrete" located at the resource given in 1 contain the implementation and are inherited from the specification classes
3. Provability of the classes: this is the subject to further investigation
4. Continuity of the solution: since Eiffel artifacts used in the formalizations of the requirements items correspond to their natural language counterparts directly, it is visible right away how a change in one representation will affect the second one

4.2 Scalability

A formal representation of a requirements item specified with Eiffel is as big as the scope of the item and its natural language description are, so the overall complexity of the final document should not depend on the size of the project. Anyway, this is something to test by applying the method to a bigger project.

4.3 Future Work

The next steps include:

1. To formally prove that the specification is consistent. In particular to ensure that the features specifications preserve what is stated in the invariants; to ensure that the expressions stated in the invariants are consistent between each other: for example it should not be possible for $P(x)$ and $\neg P(x)$ to hold at the same time
2. To formally prove that the implementation actually satisfies the features specifications
3. To extend BON notation [15] so that it would be capable of expressing model-based contracts
4. To design machinery for translating model-based contract-oriented requirements to their natural language counterpart so that the result would be recognizable by a human being.

5. To apply the method to a bigger project

The AutoProof technology [14] may be utilized for automating the aforementioned proofs. AutoProof is already capable of proving that a feature implementation preserves its specification (the postcondition holds after the feature invocation assuming the precondition), and it should be empowered with the capabilities for working solely on the specifications level so that completing the goal 1 will be possible.

As a result of implementing the aforementioned plans a powerful framework for expressing all possible views on the software under construction should emerge.

5 Acknowledgment

The author would like to thank his colleagues at the Innopolis University Software Engineering Laboratory for their invaluable feedback and guidance: Dr. Bertrand Meyer, Dr. Victor Rivera, Alexander Chichigin, Dr. Manuel Mazzara.

References

[1]. Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[2]. R Gmehlich, K Grau, M Jackson, C Jones, F Loesch, and M Mazzara. Towards a formalism-based toolkit for automotive applications. 2012.

[3]. Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[4]. Michael Jackson. The world and the machine. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 283–283. IEEE, 1995.

[5]. Michael Jackson. *Problem frames: analysing and structuring software development problems*. Addison-Wesley, 2001.

[6]. Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *Proceedings of the 17th international conference on Software engineering*, pages 15–24. ACM, 1995.

[7]. Nancy G Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *Software Engineering, IEEE Transactions on*, 20(9):684–707, 1994.

[8]. V´ıctor Rivera and N. Catan˜o. Translating Event-B to JML-Specified Java programs. In *29th ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*, Gyeongju, Korea, March 24-28 2014.

[9]. Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[10]. Bertrand Meyer. *Touch of Class: learning to program well with objects and contracts*. Springer, 2009.

[11]. Bertrand Meyer. Multirequirements. *Modelling and Quality in Requirements Engineering (Martin Glinz Festscrhift)*, 2013.

[12]. Alexandr Naumchev, Bertrand Meyer, and Victor Rivera. Unifying requirements and code: an example. The work is not published.

[13]. Nadia Polikarpova. *Specified and verified reusable components*. PhD thesis, Diss., Eidgenö¨ssische Technische Hochschule ETH Zu¨rich, Nr. 21939, 2014, 2014.
[14]. Julian Tschannen, Carlo A Furia, Martin Nordio, and Nadia Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. *arXiv preprint arXiv:1501.03063*, 2015.
[15]. Kim Wald´en and Jean Marc Nerson. *Seamless object-oriented software architecture*. Prentice-Hall, 1995.
[16]. Bertrand Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

Бесшовная разработка программного обеспечения: применимость на примере

Александр Наумчев <a.naumchev@innopolis.ru>,
Университет Иннополис,
г. Иннополис, Российская Федерация

Аннотация. В рамках традиционной программной инженерии требования и код развиваются в двух параллельных мирах. Обычная точка зрения на программную инженерию рассматривает требования и исходный код как разные артефакты, за которые несут ответственность разные люди. Этот подход, однако, влечет накладные расходы на коммуникацию и порождает проблему поддержания консистентности различных артефактов в случае необходимости внесения изменений в один из них. Изменение, внесенное в один из упомянутых артефактов, необходимо синхронизировать с остальными артефактами. В определенный момент ситуация неизбежно выходит из-под контроля: например, в случае обнаружения критического дефекта во время эксплуатации разработчики без промедления приступают к исправлению дефекта, поскольку в такой ситуации нет времени ждать, пока системные аналитики и архитекторы обновят свои документы, позволив разработчикам внести нужные изменения в код. Проблема частично решается сложными системами управления версиями, которые дороги в обслуживании и требуют соответствующей квалификации обслуживающего технического персонала. Возможно ли объединить миры требований и кода? Такое объединение упростило бы изменение и повторное использование программного обеспечения. Целесообразность применения нового подхода нуждается в изучении. В представленном исследовании рассмотрен классический пример из литературы в области проектирования требований. Для спецификации предметной области, равно как и конечного программного решения, использована одна и та же нотация – язык программирования. Данная работа содержит описание подхода, а также оценку его преимуществ, возможных ограничений и масштабируемости.

Keywords: software engineering; requirements specifications; multirequirements; Eiffel

DOI: 10.15514/ISPRAS-2015-27(3)-4

Для цитирования: Наумчев Александр. Бесшовная разработка программного обеспечения: применимость на примере. Труды ИСП РАН, том 27, вып. 3, 2015 г., стр. 57-72 (на английском языке). DOI: 10.15514/ISPRAS-2015-27(3)-4.

Список литературы

[1]. Jean-Raymond Abrial. Modeling in Event-B: system and software engineering. Cambridge University Press, 2010.

[2]. R Gmehlich, K Grau, M Jackson, C Jones, F Loesch, and M Mazzara. Towards a formalism-based toolkit for automotive applications. 2012.

[3]. Charles Antony Richard Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580, 1969.

[4]. Michael Jackson. The world and the machine. In Software Engineering, 1995. ICSE 1995. 17th International Conference on, pages 283–283. IEEE, 1995.

[5]. Michael Jackson. Problem frames: analysing and structuring software development problems. Addison-Wesley, 2001.

[6]. Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In Proceedings of the 17th international conference on Software engineering, pages 15–24. ACM, 1995.

[7]. Nancy G Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. Software Engineering, IEEE Transactions on, 20(9):684–707, 1994.

[8]. V´ıctor Rivera and N. Catan˜o. Translating Event-B to JML-Specified Java programs. In 29th ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT), Gyeongju, Korea, March 24-28 2014.

[9]. Bertrand Meyer. Object-oriented software construction, volume 2. Prentice hall New York, 1988.

[10]. Bertrand Meyer. Touch of Class: learning to program well with objects and contracts. Springer, 2009.

[11]. Bertrand Meyer. Multirequirements. Modelling and Quality in Requirements Engineering (Martin Glinz Festscrhift), 2013.

[12]. Alexandr Naumchev, Bertrand Meyer, and Victor Rivera. Unifying requirements and code: an example. The work is not published.

[13]. Nadia Polikarpova. Specified and verified reusable components. PhD thesis, Diss., Eidgeno¨ssische Technische Hochschule ETH Zu¨rich, Nr. 21939, 2014, 2014.

[14]. Julian Tschannen, Carlo A Furia, Martin Nordio, and Nadia Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. arXiv preprint arXiv:1501.03063, 2015.

[15]. Kim Wald´en and Jean Marc Nerson. Seamless object-oriented software architecture. Prentice-Hall, 1995.

[16]. Bertrand Meyer. Eiffel: A language and environment for software engineering. Journal of Systems and Software, 8(3):199–246, 1988.