A Framework for the Semantics of Behavioral Contracts

Ashley McNeile

Metamaxim Ltd, 48 Brunswick Gardens, London W8 4AN, UK
ashley.mcneile@metamaxim.com

Abstract. Contracts have proved a powerful concept in software engineering, to the point where we have a well regarded software development approach centered on their use, and native support in programming languages. However, it is not clear that there is a uniform concept of contract that can be applied to both algorithmic and interactive software. We propose a framework for thinking about software contracts based on the idea that a contract aims to preserve a formal proposition about the software system being built, and the form of the contract must reflect both the proposition whose truth is to be preserved and the formal reasoning system used for verification. In particular, we believe that very different forms of contract are needed for algorithmic and interactive software and illustrate this with examples.

1 Introduction

The idea of using contracts in the design and testing of software is well established, particularly because of the work of Bertrand Meyer in promoting *Design by Contract* (DbC) [3]. DbC is primarily about ensuring the correctness of the design of *algorithms*, where an algorithm is given a starting state (which can be thought of as the input to the algorithm) and executes to an ending state (which can be thought of as the output from the algorithm). A simple DbC contract is specified in terms of conditions on the starting and ending states termed *pre-* and *post-conditions*. An algorithm satisfying the contract will guarantee that, provided the starting state meets the pre-conditions, the ending state will meet the post-conditions. This is a simple yet powerful concept. For example, the contract for a "sort" algorithm might be specified as follows:

– The pre-condition is that a finite list of items is provided at the start, and the items in the list support a collating operator whereby any two items in the list can be compared to determine if they are equal or, if not, which is greater.
– The post-condition is that the ending state contains the same items as the start state, but arranged in non-descending sequence by the collating operator.

This completely and formally defines what is required of the sort algorithm, independently of the construction or selection of an algorithm to implement it.

DbC works well here because we are able to define the result of the sort without needing to specify or constrain in any way the states the software may take up in achieving the result.

The concepts and terminology of contracts: *pre-conditions*, *post-conditions*, *assertions* and *invariants* have now become part of the currency of software engineering, and many programming languages provide support for DbC.[1] DbC is promoted as a way of constructing programs that ensures that they are correct by design, and the concepts of DbC have spread into other areas of software engineering, such as the definition of the semantics of modeling languages (see McNeile and Roubtsova [2]).

1.1 Two Paradigms of Computing

It is tempting to suppose that the ideas of DbC are universal in computing and can be carried across seamlessly to the world of interactive software. But this world is different. In a lecture celebrating the work of Alan Turing, Milner describes the worlds of algorithm and interaction as the *OLD COMPUTING* and *NEW COMPUTING* [8] and advocates the idea that the two should be thought of as different paradigms of computing. Interaction involves *protocol* as well as *computation* and this complicates the idea of contractual conformance: If an interactive software component must obey a certain protocol in communicating with its environment, how do you specify the protocol in the contract? Is it enough that interactive software observes the correct protocol if this ensures that it is compatible with its environment; or must its contract specify the computation it performs as well as its protocol? These are not straightforward questions, and in this paper we try and provide a way of thinking about them.

1.2 A Framework for Contract Semantics

Our thesis is that the semantics of contracts must be defined with reference to a *proposition* about the software system as whole that satisfaction of the contract will ensure is true. Based on this idea, we propose a *framework* for thinking about contracts, as follows. For a given contract C we require:

- A formal system of reasoning (or theory), $\mathcal{T}$.
- A proposition, $\mathcal{P}$ articulated in $\mathcal{T}$, that we require to be true of our software as a whole.
- A *partial* proof, E articulated in the reasoning system $\mathcal{T}$, in which we can embed C to form a *complete* proof, $E + C$, of the proposition $\mathcal{P}$.

The idea is that the contract C is a component of the proof of the proposition $\mathcal{P}$. By ensuring that a software implementation, S, meets C we ensure that, provided the S is embedded in a software environment that meets E, $\mathcal{P}$ is met by the system as a whole.

[1] The entry for Design by Contract in Wikipedia lists about a dozen languages with native support and numerous products that provide non-native support for DbC.

The central point is that the framework defines what we are trying to achieve with the contract. The framework says nothing about how we determine whether or not S meets C, it only defines what follows if S **does** meet C. In particular, it defines what we mean by "substitutability": if another component S' also meets C then we know that S' also ensures $\mathcal{P}$. So if P is our measure of correctness of the software system as a whole, we are safe to use S' instead of S.

2 Application of the Framework

We look at application of this framework to DbC and then consider its application to interactive software.

2.1 Application to DbC

DbC has its origins in Hoare Logic [4], a system for formal reasoning about the correctness of algorithms. Hoare Logic is used to establish one of two forms of correctness in an algorithm:

– *Partial Correctness*, which simply requires that if an answer is returned by the algorithm it will be correct.
– *Total Correctness*, which additionally requires that the algorithm terminates.

If we are to base a contract framework on Hoare Logic (as DbC is based), then these forms of correctness are the propositions that can be established. We can therefore think about the semantics of contracts in DbC by equating as follows:

– $\mathcal{T}$ is Hoare Logic.
– $\mathcal{P}$ is a statement of correctness (partial or total) of the algorithm.
– $E + C$ is the Hoare Logic proof of $\mathcal{P}$.
– C is a contract specified in DbC form.
– $+$ denotes procedure invocation.

So the notion here is that S is a procedure or subroutine invoked within an implementation of the algorithm. The proof $E+C$ is an argument in Hoare Logic showing that the algorithm as a whole meets $\mathcal{P}$. So if S meets C the algorithm works, and we don't care about the implementation detail of S.

It may be that E itself is composed of other contracts, so $E = C1+C2+C3+\ldots$; and it can be recursively decomposed, so $C1 = C11 + C12 + C13\ldots$. Exploiting this, of course, is the idea of the DbC strategy for software development. However, this recursive structure is a property of Hoare Logic and procedural algorithms, and not inherent in the contract framework we propose.

2.2 Application to Interactive Software

The key to thinking about contracts for interactive behavior is to note that we are generally not concerned, or not only concerned, with whether or not a correctly computed answer is returned. Think of a computer chess game. If you play chess against software you will expect that it interacts with you correctly: it makes one move at a time and the moves it makes are legal according to the rules of chess. The software will assume that its opponent does likewise, otherwise the whole interaction is meaningless. But it is nonsense to talk about the interaction giving the "correct answer" - unless it is that you win the game!

Consider a player, D, who is completely deterministic in that there is only one move that D will make for a given configuration of the board; and a chess game G against whom D plays. We can ask: What proposition we would like to be true of the system $D+G$, where $+$ represents "plays against", for the purposes of considering what contract the game G should satisfy? Some possibilities are:

1. $D+G$ always executes a valid game of chess, and is deterministic so there are only two distinct games that ever get played (which occurs being determined by who has the first move).
2. As above, except that $D + G$ is non-deterministic so that many different games are possible.
3. As above, except that D always wins.

Of these, perhaps (2) is the most likely. In in terms of substitutability of G, the second would mean that, having replaced G with G', you still be able to play chess, but the games would be different and, perhaps, better. It is often the case, as it is here, that the protocol is important but the computation is not: so the algorithm by which the chess game computes its next move from the disposition of pieces on the board does not form part of the contract. This is in contrast with classical DbC, which is entirely concerned with the results of algorithmic computation.

The general implication of our framework is that, if you wish to use contracts as part of a software engineering process, you have decide on the *proposition* whose truth you want the contract to help enforce, and you have to decide on the *reasoning system* you are going to use as the basis for verification. These will then determine the form and nature of the contract you should use. In the context of procedural algorithms, Hoare Logic and the correctness propositions it supports are the only obvious choice. In the context of interactive software, because of the complexities of interplay between computation and protocol, there are other choices that can be made. The remainder of this paper discusses an example to illustrate this.

3 Example

In this section we describe *Protocol Contracts*, an example that illustrates the use contracts in the context of interactive software.

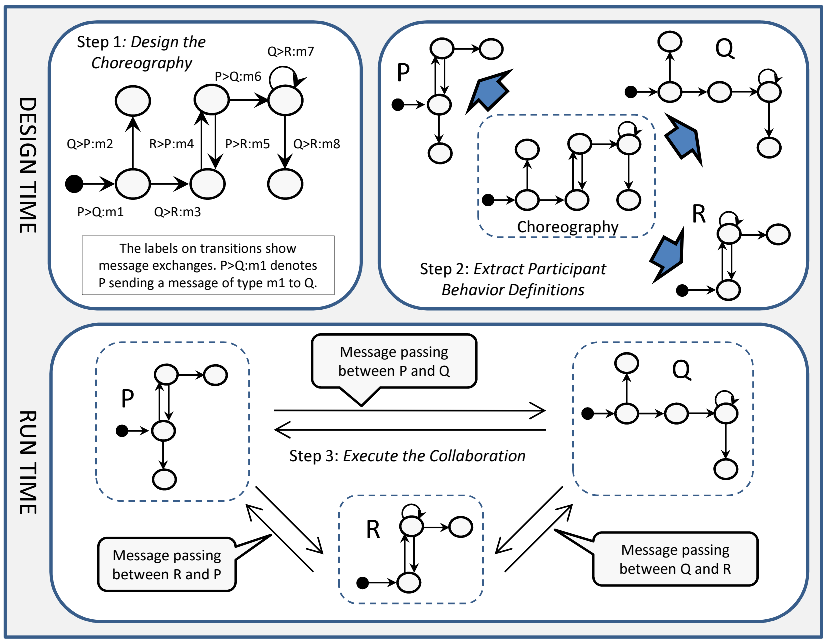
DESIGN TIME
Step 1: Design the Choreography
Q>R:m7
P>Q:m6
Q>P:m2
R>P:m4
P>R:m5
Q>R:m8
P>Q:m1
Q>R:m3
The labels on transitions show message exchanges. P>Q:m1 denotes P sending a message of type m1 to Q.
P
Q
R
Choreography
Step 2: Extract Participant Behavior Definitions
RUN TIME
Message passing between P and Q
Step 3: Execute the Collaboration
Message passing between R and P
Message passing between Q and R

Fig. 1. Extracting Participant Designs from a Choreography

Protocol contracts are a form of behavioral contract, introduced by the author in the context of designing and verifying collaborations between multiple participants using a *choreography* based approach [1]. Building a collaboration based on a choreography has three steps as depicted in Figure 1:

1. At design-time, a choreography is developed which describes all meaningful sequences of message exchange between the participants.
2. Using some kind of mechanical process, a contract for the behavior of each participant is extracted from the overall choreography.[2]
3. At run time the participants interact, each behaving according to its own behavior contract.

At run time (step 3) the participants behave independently. Each is free to send and receive messages according to its own behavioral specification and without any central orchestrating or controlling entity to enforce conformance to the original choreography. Unless the choreography obeys certain well-formedness rules, the collaboration may deadlock (leaving messages unprocessed), or may allow patterns of message exchange not envisaged in the original choreography resulting in combinations of states of the participants that were not intended and have no meaning. Determining the required well-formedness rules that mean that the collaboration is **bound to adhere** to the choreography is known as *the real-*

[2] In the jargon of choreography theory this is called “end point projection”.

izability problem. The details of the problem are beyond the scope of this paper, but the form of the contract used in steps 2 and 3 is instructive as an example.

The contract framework is as follows:

- $\mathcal{T}$ is a combination of trace based and topological reasoning, based on:
  - Properties of C. Hoare's process algebra CSP [5].
  - Rules that define well-formedness of a choreography.
- $\mathcal{P}$ is a statement that the choreography must be realizable. This means that:
  - The sequencing of message sends in the collaboration must follow the choreography, and
  - Every message sent must eventually be received and processed.
- $E + C$ is a proof of realizability.
- C is a contract specified in protocol contract form.
- $+$ denotes composition with other participants in an asynchronous collaboration.

Contracts are defined using a device called a *protocol machine*: a deterministic machine with a defined alphabet of *actions* (message sends and receives) which it either allows or refuses based on its state. A protocol contract $\mathbb{C}$ for the behavior of a participant in the collaboration is defined as a pair $[C, F]$ where:

- C is a protocol machine that defines the legal sequences of actions allowed in the participant.
- F is a set of actions of the participant that are *fully constrained* by $\mathbb{C}$.

A design, M, for the participant satisfies the contract $\mathbb{C}$ iff firstly:

$$M \parallel C = M \tag{1}$$

where $\parallel$ denotes the parallel composition operator of CSP, and secondly:

$$\begin{aligned} \forall\ a \in F\ \exists\ X \text{ such that } M = X \parallel C \text{ and} \\ X \text{ cannot refuse actions of type } a \end{aligned} \tag{2}$$

The first condition, (1), has two implications:

- The alphabet of M is a superset of that of C. This is natural, as you would not expect a design to satisfy a contract if it does not have in its alphabet all actions required by (in the alphabet of) the contract.
- Every trace of M, when restricted by eliminating those actions not in the alphabet of C, must be a trace of C. This is sometimes described as M being *Observationally Consistent*[3] with C.

The second condition, (2), states that the acceptability of any action which is *fully constrained* by the contract is determined by the state of the contract alone. In the condition, as X cannot refuse a, whether a is acceptable is determined completely by C. In other words, for an action in F, acceptability by the contract

[3] As defined by Ebert and Engels in [6].

implies acceptability by M. If an action is *not fully constrained* by the contract, acceptability by the contract is a necessary **but not a sufficient** condition for acceptability by M.

The motivation for using this contract based approach for the design of participants is that:

– The choreography does not completely specify the behavior of a participant. This is supported by the use of protocol machines, which specify *possible* orderings of actions but leave the *choice* of action in a given state open.
– A participant can simultaneously engage in two or more completely independent choreographies and, in this case, each choreography gives rise to a contract for the participant. This is handled by the fact that protocol contracts can be composed, the composition of two contracts being another contract.
– Realizability of a choreography for asynchronous collaboration requires that receive actions in a participant must not be prevented (refused). For instance, suppose that a customer requests a loan from a bank. The bank may either grant or deny the loan, at its own discretion. But the customer has no discretion about choosing which message to receive: it must receive whatever the bank sends. This is specified in the contract using the mechanism for fully constrained actions, F.

Finally, note that a contract is concerned with protocol (that a loan once requested is either granted or denied) and not with computation. How the bank decides whether to grant or deny a loan is of no consequence for realizability of the choreography, and two banks that both obey the collaboration contract are substitutable for each other with respect to realizability even if the processes and criteria they use for deciding when to grant and when to deny credit are completely different.

4 Contracts and Specifications

It might be supposed that no distinction can be made between the concept of a *contract* and that of a *specification.* To some extent this depends on how the words are used. However we suggest that the two can be distinguished as different concepts serving different purposes.

A specification can be characterized as *an instrument of delegation or procurement.* This characterization (the author's own) reflects the role that a specification plays as a contract between a *supplier* and *customer* in the context of a procurement, in particular as the ultimate determinant used by both parties of what constitutes a satisfactory delivery. The contract here is between human agents and quite different from the concept of a software contract. It follows that there are some differences between a (software) specification and a (software) contract.

Firstly, a software contract is not (normally) a complete specification. Consider two of the examples used in this paper:

– In the *sort* example given in Section 1, the DbC contract is purely functional. However a specification for a sort algorithm would generally require some definition of non-functional requirements, such as constraints on the size of list that can be sorted and the required performance of the algorithm using given computing resources.
– In the *bank lending* example given at the end of Section 3, the protocol contract makes no constraint on how the decision to grant or deny a loan is made. However, if a bank were procuring this software (perhaps from its internal IT department) it would need to specify the algorithm that makes this decision very carefully.

In other words, to serve as a basis for procurement a specification must be *complete* in a way that a contract need not be.

Secondly, a given piece of software may be subject to multiple contracts. Thus, for instance, a participant in a collaborative computing environment may engage simultaneously in more than one separately choreographed collaboration. Each choreography imparts a contract for the participant, and the participant must comply with all them simultaneously. This is in contrast with a specification, where uniqueness is essential to ensure that customer and supplier have the same shared understanding of what it to be delivered, and it is the specification that should embody this shared understanding. In this sense, a specification must be *unique* in a way that a contract need not be.

5 Modeling Semantics

Using these ideas could help to sharpen work on defining the semantics of modeling languages and notations. Consider the specification of the semantics of a *Protocol State Machine* in UML, part of which is stated as follows:

"The interpretation of the reception of an event in an unexpected situation (current state, state invariant, and pre-condition) is a *semantic variation point*: the event can be ignored, rejected, or deferred; an exception can be raised; or the application can stop on an error. It corresponds semantically to a pre-condition violation, for which no predefined behavior is defined in UML" [7].

This uses the language of DbC style contracts (pre-condition) but we note that:

– There is no consideration of what proposition a Protocol State Machine (as a contract) might be used to enforce.
– The use of DbC style contracts (which are used for algorithms) is curious as the context is a formalism that is nominally about protocol.

This may explain why the authors have resorted to using the "variation point" (i.e., undefined) construct. Without identifying the proposition (or propositions, as there may be a choice) to be enforced, the definition of semantics will inevitably lack clarity. In our view the contract framework discussed here could be used to improve this part of the UML specification.

6 Conclusion

In this paper we have tried to show that the concept of *contract* in software should be viewed as a general concept, of which the familiar DbC style, using pre- and post-conditions, is an example. We propose a framework that allows a more general consideration, based on the idea that we start with a proposition which we are trying to ensure is met, and design contracts based on the reasoning system in which this proposition can be proved. We suggest that this framework can help to widen the scope and use of contracts in software design, and serve to sharpen the definition of semantics of modeling languages.

Bibliography

[1] A. McNeile. Protocol Contracts with application to Choreographed Multiparty Collaborations. *Service Oriented Computing and Applications*, 2010. doi: http://dx.doi.org/10.1007/s11761-010-0060-9.

[2] A. McNeile and E. Roubtsova. Composition Semantics For Executable and Evolvable Behavioral Modeling in MDA. In *BM-MDA '09: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 1–8, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-503-1. doi: http://doi.acm.org/10.1145/1555852.1555855.

[3] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000. ISBN 0-13-629155-4.

[4] C. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/363235.363259.

[5] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985. ISBN 0-13-153271-5.

[6] J. Ebert and G. Engels. Observable or Invocable Behaviour - You have to choose. *Technical report 94-38. Department of Computer Science, Leiden University*, 1994.

[7] OMG. Unified Modeling Language, Superstructure, v2.2. *OMG Document formal/09-02-02 Minor revision to UML, v2.1.2. Supersedes formal 2007-11-02*, 2009.

[8] R. Milner. Turing, Computing and Communication. In *Lecture for the 60'th anniversary of Turing's "Entscheidungsproblem"*. King's College, Cambridge, England, 1997.