Design By Contract
Deontic Design Language for Component-Based Systems

Christophe Garion[a] Leendert van der Torre[b]

[a] SUPAERO, Toulouse, France
[b] CWI, Amsterdam, the Netherlands

Abstract

Design by contract is a well known theory that views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by assertions. However, there is a gap between this theory and software engineering concepts and tools. For example, dealing with contract violations is realized by exception handlers, whereas it has been observed in the area of deontic logic in computer science that violations and exceptions are distinct concepts that should not be confused. To bridge this gap, we propose a software design language based on temporal deontic logic. We also discuss the relation between the normative stance toward systems implicit in the design by contract approach and the intentional or BDI stance popular in agent theory.

1 Introduction

Design by contract [10, 11, 12] is a well known software design methodology that views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by assertions. It has been developed in the context of object oriented programming, it is the basis of the programming language Eiffel, and it is well suited to design component-based and agent systems. However, there is still a gap between this methodology and formal tools supporting it. For example, dealing with contract violations is realized by exception handlers, whereas it is well known in the area of deontic logic in computer science [13, 18] that violations and exceptions are distinct concepts that should not be confused. Formal tool support for design by contract is therefore a promising new application of deontic logic in computer science [19]. In this paper we study how deontic logic can be used as a design language to support design by contract. We address the following three research questions.

1. Which kind of deontic logic can be used as a design language to support design by contract?
2. What kind of properties can be formalized by such a design logic?
3. How does this approach based on deontic logic compare to the BDI approach, dominant in agent based software engineering?

The motivation of our work is the formal support for agent based systems. Recently several agent languages and architectures have been proposed which are based on obligations and other normative concepts instead (or in addition to) knowledge and goals, or beliefs, desires and intentions. In artificial intelligence the best known of these normative approaches is probably the IMPACT system developed by Subrahmanian and collegues [6]. In this approach, wrappers built around legacy systems are based on obligations. However, we are interested in particular in component based agent systems such as the BOID architecture [3].

The layout of this paper is as follows. In Section 2 we discuss design by contract. In Section 3 we discuss contract violations. In section 4 we compare this approach based on deontic logic to the BDI approach.

2 Design by contract

We explain design by contract by an example program in the Eiffel programming language. The explanation of design by contract as well as the example have been taken from [9]. For further details on design by contract, see [10, 11, 12].

Design By Contract views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by *assertions*. These assertions play a central part in the Eiffel method for building reliable object-oriented software. They serve to make explicit the assumptions on which programmers rely when they write software elements that they believe are correct. In particular, writing assertions amounts to spelling out the terms of the *contract* which governs the relationship between a routine and its callers. The precondition binds the callers; the postcondition binds the routine.

The Eiffel class in the left column of Figure 1 illustrates assertions (ignore for now the right column). An account has a balance (an integer) and an owner (a person). The only routines – **is** ... **do** ... **end** sequences – accessible from the outside are increasing the balance (deposit) and decreasing the balance (withdraw). Assertions play the following roles in this example.

Routine preconditions express the requirements that clients must satisfy when they call a routine. For example the designer of *ACCOUNT* may wish to permit a withdrawal operation only if it keeps the account's balance at or above the minimum. Preconditions are introduced by the keyword **require**.

Routine postconditions, introduced by the keyword **ensure**, express conditions that the routine (the supplier) guarantees on return, if the precondition was satisfied on entry.

A class invariant must be satisfied by every instance of the class whenever the instance is externally accessible: after creation, and after any call to an exported routine of the class. The invariant appears in a clause introduce by the keyword **invariant**, and represents a general consistency constraint imposed on all routines of the class.

Syntactically, assertions are boolean expressions. To formalize the assertions in our design language, we use a deontic logic based on directed obligations, as

```
class ACCOUNT
feature
 balance: INTEGER
 owner: PERSON
 min_balance: INTEGER is 1000
 deposit(sum:INTEGER) is
  require
   sum >= 0                          Ocr(sum >= 0)
  do
   add(sum)
  ensure
   balance = old balance + sum       Orc(balance = old balance + sum)
  end
 withdraw(sum:integer) is
  require
   sum >= 0                          Ocr(sum >= 0)
   sum <= balance - min_balance      Ocr(sum <= balance - min_balance)
  do
   add(-sum)
  ensure
   balance = old balance - sum       Orc(balance = old balance - sum)
  end
feature [NONE]
 add(sum:INTEGER) is
  do
   balance:=balance+sum
  end
invariant
 balance >= min_balance             Or(balance >= min_balance)
end -- class ACCOUNT
```

Figure 1: class $ACCOUNT$

used in electronic commerce and in artificial intelligence and law [5, 7, 15, 17]. A modal formula $O_{ab}(\phi)$ for a, b in the set of objects (or components, or agents) is read as "object a is obliged toward object b to see to it that ϕ holds". We write c and r for the caller and for the routine, such that the assertions in the program can be expressed as the logical formulae given in the right column in Figure 1. Summarizing:

Require ϕ	=	$O_{cr}(\phi)$:	caller is obliged toward routine to see to ϕ.
Ensure ϕ	=	$O_{rc}(\phi)$:	routine is obliged toward caller to see to ϕ.
Invariant ϕ	=	$O_{r}(\phi)$:	routine is obliged to see to ϕ.

To use these obligations to a deontic design language, we have to add temporal

information. First, we have to formalize **old** *expression*, which is only valid in a routine postcondition. It denotes the value the expression has on routine entry. Consequently, we have to distinguish between expressions true at entry of the routine and at exit of it. More generally, we have to reason how the assertions change over time. For example, the require obligation only holds on entrance, the ensure obligation holds on exit, and the invariant obligation holds as long as the object exists. The obligations only hold conditionally. For example, if the preconditions do not hold, than the routine is not obliged to see to it that the ensure expression holds. Finally, the conditional obligations come into force once the object is created, and cease to exist when the object is destructed.

We therefore combine the logic of directed obligations with linear time logic (LTL), well known in specification and verification [8]. There are many alternative temporal logics which we could use as well. For example, in [4] deontic logic is extended with computational tree logic in $\mathrm{BDIO}_{\mathrm{CTL}}$. Semantics and proof theory are straightforward, see for example [4]. Due to space limitations, we do not give the details. Instead, we address the question how to use the logic to reason about assertions.

Definition 1 (Syntax $\mathbf{O_{LTL}}$) *Given a finite set A of objects (or components, or agents) and a countable set P of primitive proposition names. The admissible formulae of O_{LTL} are recursively defined by:*

1 Each primitive proposition in P is a formula.

2 If α and β are formulae, then so are $\alpha \wedge \beta$ and $\neg\alpha$.

3 If α is a formula and $a, b \in A$, then $O_{a,b}(\alpha)$ is a formula as well.

4 If α and β are formulae, then $X\alpha$ and $\alpha U \beta$ are formulae as well.

We assume the following abbreviations:

$$\begin{array}{lllllll} \alpha \vee \beta & \equiv_{def} & \neg(\neg\alpha \wedge \neg\beta) & \qquad & \alpha \rightarrow \beta & \equiv_{def} & \neg\alpha \vee \beta \\ \Diamond(\alpha) & \equiv_{def} & \top U \alpha & & \Box(\alpha) & \equiv_{def} & \neg\Diamond(\neg\alpha) \\ O_a(\alpha) & \equiv_{def} & O_{a,a}(\alpha) & & & & \end{array}$$

We assume the following propositions: create(c) holds when object c is created, destruct(c) holds when object c is destructed, call(c_1,c_2,f) holds when object c_1 calls routine f in object c_2. We assume that if a routine in an object is called, there is an earlier moment in time at which the object is created. However, since our operators only consider the future, this property cannot be formalized. We assume that propositions can deal with integers, a well known issue in specification and verification, see [8] for further details. Finally, we assume that the time steps of the temporal model are calls to routines. The first routine and the invariant in the example can now be formalized as:

$$\begin{array}{l} \text{call}(c_1,c_2,\text{deposit(sum:INTEGER)}) \rightarrow O_{c_1,c_2}(\text{sum} >= 0) \\ (\text{call}(c_1,c_2,\text{deposit(sum:INTEGER)}) \wedge (\text{sum} >= 0) \wedge (\text{balance} = \text{b})) \rightarrow \\ \qquad\qquad XO_{c_2,c_1}(\text{balance} = \text{b} + \text{sum}) \\ \text{create}(c) \rightarrow (O_c(\text{balance} >= \text{min\_balance}) \text{ U } \text{destruct(c)}) \end{array}$$

3 Contract violations

Whenever there is a contract, the risk exists that someone will break it. This is where exceptions come in. Exceptions – contract violations – may arise from several causes. One is an assertion violation, if run-time assertion monitoring is selected. Another is a signal triggered by the hardware or operating system to indicate an abnormal condition such as arithmetic overflow, or an attempt to create a new object when there is no enough memory available. Unless a routine has been specified to handle exceptions, it will **fail** if an exception arises during its execution. This in turn provides one more source of exceptions: a routine that fails triggers an exception in its caller.

A routine may, however, handle an exception through a **rescue** clause. An example using the exception mechanism is the routine *attempt_deposit* in Figure 2 that tries to add sum to balance. The actual addition is performed by an external, low-level routine *add*; once started, however, *add* may abruptly fail, triggering an exception. Routine *attempt_deposit* tries the deposit at most 50 times; before returning to its caller, it sets a boolean attribute *successful* to *True* or *False* depending on the outcome. This example illustrates the simplicity of the mechanism: the **rescue** clause never attempts to achieve the routine's original intent; this is the sole responsibility of the body (the **do** clause). The only role of the **rescue** clause is to clean up the objects involved, and then either to fail or to retry.

```
attempt_deposit(sum:INTEGER) is
 local
  failures: INTEGER
 require
  sum >= 0;                          Ocr(sum >= 0)
 do
  if failures < 50 then
   add(sum); successful := True
  else
   successful := False
 rescue
  failures := failures + 1; retry
 ensure
  balance = old balance + sum        Orc(balance = old balance + sum)
 end
```

Figure 2: Routine *attempt_deposit*

The principle is that *a routine must either succeed or fail*: it either fulfills its contract, or not; in the latter case it must notify its caller by triggering an exception. The optional **rescue** clause attempts to "patch things up" by bringing the current object to a stable state (one satisfying the class invariant). Then it can terminate in either of two ways:

- The **rescue** clause may execute a **retry** instruction, which causes the rou-

tine to restart its execution from the beginning, attempting again to fulfil its contract, usually through another strategy. This assumes that the instructions of the **rescue** clause, before the retry, have attempted to correct the cause of the exception.

- If the **rescue** clause does not end with a **retry**, then the routine fails; it returns to its caller, immediately triggering an exception. (The caller's **rescue** clause will be executed according to the same rules.)

In our design language, the exception can be formalized as a violation, and the exception handler gives rise to a so-called contrary-to-duty obligation. For example, there is a violation if the postcondition does not hold, i.e., we do not have balance = old balance + sum. In case of violation, a retry means that the obligation persists until the next time moment. We extend the language with the proposition *retry*. Now, the fact that a retry implies that the postcondition holds again for the next moment can be characterized as follows:

$$O_{c_1,c_2}(\phi) \wedge \neg\phi \wedge \mathit{retry} \rightarrow XO_{c_1,c_2}(\phi)$$

4 The normative stance

In this section we compare the normative stance, a phrase due to Jan Broersen [2], implicit in the design by contract with the intentional or BDI stance popular in agent based software engineering. Table 1 summarizes the comparison between the intentional stance and the normative stance.

Stance	intentional stance	normative stance
Concepts	BDI	OP, rights, responsibility
from	folk psychology	ethics, law, sociology
Computer	human = angry, selfish, ...	God, master/slave, servant
Class of systems	decision making	decision making
Realization	specification and verification	components
Implementation	programming	objects, operation
specification	$\mathrm{BDI}_{\mathrm{CTL}}$	temporal deontic logic

Table 1: intentional vs normative stance

First, the intentional stance is rooted in the philosophical work of Dennett, whereas such grounding does not seem to exist for the normative stance (though there are candidates, such as [1]). The concepts from the intentional stance come from folk psychology. The normative stance borrows concepts from ethics, law or sociology. Other examples of this normative stance we mentioned in the introduction are the IMPACT system [6] and the BOID architecture [3].

Second, the success of the intentional stance is that people like to talk about their computer as a human which has beliefs and desires, which may be selfish, or which can become angry. The implicit assumption of design by contract is that designers find it useful to understand software construction in terms of contracts,

or, more generally, in terms of obligations. The success is due to the fact that humans either consider the computer as their master, which has to be obeyed, or as their slave, which has to obey orders.

Third, the intentional stance has been advocated for agent systems, which are for example autonomous and proactive. It has been used as a high level specification language, as well as low level programming language. We believe the normative stance can be used in a wider setting. In the examples we used it for low level objects. However, it is particularly useful if we use a higher abstraction level in terms of components or agents.

5 Concluding remarks

In this paper we study how deontic logic can be used as a design language to support design by contract. First, we ask which kind of deontic logic can be used as a design language to support design by contract We show how directed modal operators are capable of formalizing contracts between clients (callers) and and suppliers (routines), relying on mutual obligations and benefits made explicit by assertions. These formalisms have been developed and studied in electronic commerce and artificial intelligence and law. Moreover, we show how temporal operators can be used to formalize dynamic behavior such as contract violations.

Second, we ask what kind of properties should be formalized by such a design logic. This is summarized in Table 2. In this paper, we do not consider contract forms and contracts for testing and debugging. The contract form of a class, also called its "*short form*", serves as its interface documentation. It is obtained from the full text by removing all non-exported features and all implementation information such as **do** clauses of routines, but keeping interface information and in particular assertions. The use of these elements in our deontic design language, for example to *combine* assertions, is subject of further research.

social contract	assertions	directed obligations
violation	exception	violations
repair	exception handling	contrary-to-duty reasoning
contract form	interface	?
testing and debugging	?	?

Table 2: Bridging the gap

Third, we ask how this approach based on deontic logic compares to the BDI approach, dominant in agent based software engineering. Whereas the BDI approach is based on an attribution of mental attitudes to computer systems, design by contract is based on an attribution of deontic attitudes to systems. We suggest that the normative stance has a wider scope of applicability than the intentional stance, though this has to be verified in practice. In further research we study the relation with commitments in Shoham's Agent Oriented Programming (AOP) [14], and with rely/guarantee reasoning [16].

References

[1] R. Brandom. *Making it explicit*. Harvard University Press, Cambridge, MA, 1994.

[2] J. Broersen. *Modal Action Logics for Reasoning about Reactive Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2003.

[3] J. Broersen, M. Dastani, J. Hulstijn, and L. van der Torre. Goal generation in the BOID architecture. *Cognitive Science Quarterly*, 2(3-4):428–447, 2002.

[4] J. Broersen, M. Dastani, and L. van der Torre. BDIO_{CTL}: Properties of obligation in agent specification languages. In *Proceedings of IJCAI'03*, pages 1389–1390, 2003.

[5] F. Dignum. Autonomous agents with norms. *Artificial Intelligence and Law*, 7(1):69–79, 1999.

[6] T. Eiter, V. Subrahmanian, and G. Pick. Heterogeneous active agents, I: Semantics. *Artificial Intelligence*, 108:179–255, 1999.

[7] C. Krogh and H. Herrestad. Hohfeld in cyberspace and other applications of normative reasoning in agent technology. *Artificial Intelligence and Law*, 7(1):81–96, 1999.

[8] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Heidelberg, Germany, 1992.

[9] B. Meyer. Invitation to Eiffel. Technical Report TR-EI-67/IV, Interactive Software Engineering, 1987.

[10] B. Meyer. Design by contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice-Hall, New York, London, 1991.

[11] B. Meyer. Applying design by contract. *IEEE COMPUTER*, 25(10):40–51, 1992.

[12] B. Meyer. Systematic concurrent object-oriented programming. *Communication of the ACM*, 36(9):56–80, 1993.

[13] J. Meyer and R. Wieringa. *Deontic Logic in Computer Science: Normative System Specification*. John Wiley and Sons, 1993.

[14] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[15] M. P. Singh. An ontology for commitments in multiagent systems: toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.

[16] E. W. Stark. A proof technique for rely/guarantee properties. In *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391, 1985.

[17] Y. Tan and W. Thoen. Modeling directed obligations and permissions in trade contracts. In *Proceedings of the Thirty-First Annual Hawaian International Conference on System Sciences*, 1998.

[18] G.H. von Wright. Deontic logic. *Mind*, 60:1–15, 1951.

[19] R. Wieringa and J. Meyer. Applications of deontic logic in computer science: A concise overview. In *Deontic Logic in Computer Science*, pages 17–40. John Wiley & Sons, Chichester, England, 1993.