

Specifying Consequences with Action Contracts

David Gelperin
LiveSpecs Software
dave@livespecs.com

Abstract: This paper describes a technique, action contracting, for the specification of goal-directed actions. Contracting can be used to specify consequential action rules (rather than algorithms) for any goal-directed entity (e.g., person, system, product, component, class, or object). If an entity can be effectively tested for functionality, that functionality can be contracted. Action contracts are readable by clients and writeable by most software professionals, with minimal training.

1 Introduction

If we want to specify the behavior rules governing the actions of a single software system or product, whose internal structure is unknown (i.e., black-box descriptions), the Unified Modeling Language provides state diagrams and activity diagrams. Neither diagram, in their standard form, specifies the consequences (postconditions) of actions beyond the next state entered. Rather, their transition descriptions reference the actions themselves e.g., display an invalid input message. This makes action consequences implicit and assumes that the semantics of the action are well understood.

At the system or product level, the specification issue is “what exactly does the client need and want?” What do they mean by the terms that they use and do they understand the consequences of their requests? While state and activity diagrams are effective for some clients, most require an explicit statement of consequences to precisely define, and sometimes understand, their own needs.

2 Action Semantics via Consequences

The semantics of actions (e.g., sort) can be precisely specified in at least two ways by:

- (1) consequence (e.g., the value of each item in a sorted list is less than or equal to its predecessors and greater than or equal to its successors), and
- (2) algorithm (e.g., listing the steps in a bubble sort).

Both are important –

Consequences: so that we can determine if delivered functionality achieves intended results, and

Algorithms: so that process design can be constrained or specified.

Consequences, however, provide the explicit semantics of correct behavior. Algorithms provide only implicit semantics for correctness.

Consider a recipe containing a list of ingredients, a set of preparation steps, and a picture of the result. To a skilled cook, the picture and ingredient list are sufficient to describe the visual (explicit) and taste (implicit) consequences of following the preparation steps (algorithm).

3 Consequences and Verification

Understanding consequences is essential to effective testing. Those responsible for test must understand required outcomes in order to recognize success and failure. If we are interested in automatic test design, generation and verification (i.e., evaluation of test outcomes), explicit and precise consequences are required.

Unfortunately, some consequences may be difficult to specify precisely e.g., graphic displays or variable voice responses. These “hard to describe” results indicate that while tests may be automatically designed, test outcomes must be evaluated by informed human judgment, during manual testing.

Even worse, some consequences can be precisely specified, but difficult to recognize. For example, consider a system that solves various forms of the traveling salesman problem. For example, it finds the shortest route that takes a salesperson through N cities and returns them to the starting city. The system may produce a circular route, but whether the route is optimal may be hard to determine. The system may need to provide easily checked upper and lower bounds on the correct answer, as well as a solution. In the case of “hard to recognize” results, even automatic test design may not be productive. Fortunately, many interesting systems are neither “hard to describe” nor “hard to recognize”.

In addition to supporting test, explicit consequences also increase the reviewability of a specification, assuming that the consequences are readable. Reviewability requires that the notation used to express consequences balance precision with client readability. Clients ask for functionality and they require communication that contains terms and formats that they can understand. This disqualifies the Object Constraint Language [WK99] from general use.

To increase readability, avoid unnecessary mathematical expressions. For example, use “count [is] up 1” instead of “count = count + 1”. They are both precise, but one doesn’t look like a line of code – which is a good thing.

Sometimes, mathematical expressions will be required to precisely define consequences. For example, the proceeds P at the end of Y years of a fixed investment scenario where \$ dollars are invested at a fixed yearly interest rate R, and taxes are ignored, is given by: $P = \$(1+R)^Y$. If “advanced” math is required to specify a precise result, it is likely that some client representatives will be comfortable with the needed mathematical forms and therefore able to effectively review them.

4 Consequential Specs using Contract Expressions

We begin with an example to illustrate a key component of action contracts. Then we specify the key components as well as the action contracts themselves, provide additional examples, and describe usage options and a contract development strategy.

Consider a function **DaysBetween (Date1, Date2)** defined as follows:

Constant Conditions	Post-Conditions
(Date1 invalid or Date2 invalid or Date2 before Date1)	DaysBetween (Date1, Date2) = 0
Date1 valid & Date2 valid & Date2 = Date1	DaysBetween (Date1, Date2) = 1
Date1 valid & Date2 valid & Date2 after Date1	DaysBetween (Date1, Date2) = DaysBetween (Date1, (Date2 - 1 day)) + 1

Figure 1: Vertical Action Contract for DaysBetween (Date1, Date2)

The first expression says that if the input is invalid, then the value returned by the function should be zero.

The second expression says that if the input is valid and the dates are the same, then the value returned by the function should be one.

The third expression says that if the input is valid and Date2 occurs after Date1, then the value returned by the function should be one greater than the value the function should return for the same Date1 but a Date2 that is one day earlier.

Observe that these contract expressions are not natural in the sense of natural language. You would not curl up in bed and read one for enjoyment. However, it is well known that natural language presents an insurmountable ambiguity problem when precision is a priority. Contract expressions try to strike a balance between precision and readability, but for most people, their format and mode of expression take some getting use to.

We do not give details of the contract expression language, because it is still under development.

5 Formatting Contract Expressions

Contract expressions can be formatted in several ways. In the table above, three mutually exclusive expressions (i.e., only one can occur) are grouped vertically in a table read from left to right. We can also group them horizontally in a table read from top to bottom as follows:

Constant Conditions	(Date1 invalid or Date2 invalid or Date2 before Date1)	Date1 valid & Date2 valid & Date2 = Date1	Date1 valid & Date2 valid & Date2 after Date1
Post Conditions	DaysBetween (Date1, Date2) = 0	DaysBetween (Date1, Date2) = 1	DaysBetween (Date1, Date2) = DaysBetween (Date1, (Date2 - 1 day)) + 1

Figure 2: Horizontal Action Contract for DaysBetween (Date1, Date2)

In either table format, we can add a final row or column to record conditions common to all preceding entries. An example of this appears in Section 9.2.

We can also ungroup the expressions and format each one separately as a single entry table in either table format or in outline form as illustrated below:

PreConditions

Constant Conditions
Date1 valid
& Date2 valid
& Date2 = Date1

Post-Conditions
DaysBetween (Date1, Date2) = 1

Figure 3: Format for Single Contract Expression

Additional format options will be illustrated in Section 10. Formats can be mixed in a single specification. The choice of format should depend primarily on client preference.

Before defining an action contract, we specify its major components, contract expressions and essential classes.

6 Contract Expressions

Contract expressions are consequential specifications of goal-directed actions. They provide descriptions of enabling situations and the consequences of the resulting entity actions. A contract expression is composed of three parts, PreConditions, Constant conditions, and PostConditions. Each part is a logical expression i.e., one that is True or False, in conjunctive normal form (see below).

6.1 Logical Expressions

A **simple condition** is a logical statement that contains neither “and” nor “or”, but may contain “not”. For example, “temp > 98.6”, “payment not overdue”, and “zip code is 55427” are all simple conditions.

A **conjunct** is a set of one or more simple conditions combined only with logical “or”s. For example (temp < 40 or it is raining) is a conjunct containing two simple conditions. Note that a single simple condition is a conjunct by this definition.

A logical expression is in **conjunctive normal form** (CNF) if it is a set of one or more conjuncts combined only with logical “and”s. For example, if A, B, and C are simple conditions, then [(A and B) or C] is not in CNF, while the truth value equivalent form [(A or C) and (B or C)] is in CNF. In fact, every logical expression has a truth value equivalent CNF.

6.2 Pre-conditions

Each conjunct in a PreCondition logical expression must be True prior to the corresponding action in order for that action to occur. PreCondition conjuncts must be False or Unknown afterwards – otherwise they belong in an Invariant expression.

6.3 Constant Conditions

Each conjunct in a constant logical expression must be True prior to the corresponding action in order for that action to occur. It must also be unchanged by the action i.e., be True after the action, if the action occurs correctly.

6.4 Post-conditions

Each conjunct in a PostCondition logical expression must be True after the action, if the action occurs correctly. PostCondition conjuncts must be False or Unknown beforehand – otherwise they belong in an Invariant expression.

In addition to output activity and timing, post-conditions describe changes in Domain Model objects. These changes include:

- (1) instance creation and deletion,
- (2) association formation, modification, and dissolution and
- (3) attribute value modification.

6.5 Pre-condition and Post-condition Relationships

If a PreCondition conjunct J will always be False after the action, then the negation of J will yield one or more simple conditions that are each candidate conjuncts for the action’s PostCondition logical expression. For example, if the PreCondition is [(P or Q) and R] and (P or Q) will always be False after the action, then (Not P) and (Not Q) are both candidate conjuncts for the PostCondition.

The conjuncts are only candidates because the PostCondition may already contain a stronger conjunct i.e., one that implies the candidate. For example, for a traffic light controller, a contract expression specifying the transition from green to yellow, would already have “light is yellow” as a PostCondition conjunct. “light is yellow” implies “light in not green” and is therefore a stronger PostCondition conjunct.

Correspondingly, if a PostCondition conjunct K will always be False before the action, then the negation of K will yield one or more simple conditions that are each candidate conjuncts for the action’s PreCondition logical expression.

7 Essential Classes and Objects

To understand a contract expression, a reader must understand elements of the essential class model of the domain of discourse. This model results from object-oriented analysis [La02, MO95, SM89]. For example, to communicate effectively with a client about a library management system, both parties must share an understanding of “book”, “copy”, and “borrower” along with the relationships among these classes, the class attributes, and the attribute value ranges. It is in this sense that these classes and their aspects are essential. They are essential for effective communication of requirements. These elements are also called the conceptual classes in a domain model [La02].

Note that knowledge of class methods is not essential in this sense. We are focussed on specifying entity actions that will often require the coordination of several methods, but such coordination is a design issue.

Some class specifiers use methods to algorithmically define “derived” class attributes such as object validity. We advocate the explicit specification of such attributes via the logical expressions implemented by these methods. (e.g., see Figure 11 in Section 9.3).

An action contract should contain the essential classes and objects referenced in its contract expressions to assure that the contract terminology is precise. In particular the references in the contract conditions.

8 Action Contracts

An **Action Contract** consists of three elements:

- (1) a title and optional scope statement that bounds completeness,
- (2) a set of one or more contract expressions and their associated parameters, and
- (3) an optional set of essential classes defining the terminology in the contract expressions.

For simple, well-defined behavior, a contract title can provide a sufficient scope statement.

A contract is **complete** if and only if the set of expressions is complete relative to the scope statement i.e., all situations within the scope are covered. The **correctness** and completeness of a contract should be determined by incremental client review (see Section 11).

9 A System Contract

The following is an example of an action contract for an over-simplified inventory control system.

9.1 Contract Scope

Title: Inventory Control Activities Contract

Scope: A system processing single item orders and:

- (1) rejecting unacceptable (NG) orders,
- (2) reordering items from suppliers, and
- (3) processing acceptable (OK) orders

This description identifies the system and provides an action list in which each action is described at a specific level of granularity. For example, the third action covers many situations such as processing when the inventory is adequate to fill the order and when it is not.

In any case, for each action the question remains, “What exactly does it mean for the system to perform this particular action?”.

9.2 Contract Expressions

We answer by specifying the actions of the inventory control system with a set of contract expressions. Again the granularity issue arises. We can leave an action at its original level of granularity or decompose it. The granularity of actions 1 and 2 are preserved, while action 3 is decomposed into four subactions.

By convention, contract expressions apply at most once to a single situation (e.g., single order or single item in a multi-item order).

Functions	Constant Conditions	Post-Conditions
Process NG orders	& Ord verified NG	NG order rejected & errors logged & invalid orders up 1
Reorder item	& Ord verified OK & before Packing & reorder point < onhand quantity & reorder point >= (onhand quantity – quantity of Ord)	item reorder created and filed & on-order quantity up (reorder quantity)
Create backorder	& Ord verified OK & before Packing & onhand quantity < quantity of order	backorder printed & backorder quantity up (quantity of order – onhand quantity)
Create invoice	& Ord verified OK & Cust is <customer> & before Packing & onhand quantity > 0	invoice created and filed & amount due from Cust up (total amount of order) & invoices of Cust up 1
Save OK orders	& after Packing	OK order saved & valid orders up 1
	Common Conditions: Ord is <order>	

Figure 4: Contracts for an Inventory Control System (1 of 2)

Fillable	PreConditions	Constant Conditions	Post-Conditions
Fully	onhand quantity >= quantity of Ord		& packing slip printed for full order
Partly	onhand quantity > 0	& onhand quantity < quantity of Ord	& packing slip printed for partial order
Not		& onhand quantity = 0	& no packing slip printed
		Common conditions: Ord is <order> & Ord verified OK	Common conditions: onhand quantity down (packed quantity)

Figure 5: Contracts for **Pack orders** function of ICS (2 of 2)

9.3 Essential Classes

This section would contain the essential class definitions for customers, orders, and item inventories along with definitions for reorders, backorders, invoices, and packing slips. We illustrate using essential class definitions for orders.

Class Name	Attribute Name	Attribute Description	Value Ranges	Attribute Identifier
Item Order	Order Id		Order Ids	At01
	Order Moment	Date & Time	Business Dates & Times	At02
	Order Taker Id		Order Taker Ids	At03
	Customer Id		Customer Ids	At04
	Item Id		UPC codes	At05
	Order Quantity		1 to 999	At06
	Delivery Moment	Date & Time	Business Dates & Times	At07
	Delivery Location		Location codes	At08
	Payment Type		Cash or Credit	At09

Figure 6: Basic attributes of item orders

Class Name	Derived Attribute Name	Value Name	Value Definition
Item Order	Validity		
		OK	At01 to At09 all valid
		NG	Not OK

Figure 7: Derived attributes of item orders

10 Symmetric State Expressions

None of the previous examples involve states. In order to incorporate state models into action contracts, we adjust the definition of the standard extended state transition rule resulting in a symmetric state expression.

Symmetric state expressions differ from the standard transition rules by replacing the actions with the consequential conditions resulting from those actions. In what follows, we view states as the cross-product of one or more system modes.

Symmetric state expressions (transition rules) have the form:

If preconditions (modes, triggers, & supplemental preconditions),
Then postconditions (consequential conditions, supplemental postconditions
& modes).

Supplemental pre and post conditions as well as consequential conditions are optional in a symmetric state expression. Actions or actors may be included as comments. We use the term “symmetric” to denote that these expressions have only conditional expressions on each side of the transition rule i.e., no actions.

Since action contracts incorporate a form of state model, some contract expressions can describe state transitions, while others can describe pure functional results i.e., no states. This permits more congruent modeling of real systems. See example below.

The 2-part table below illustrates the use of symmetric state tables for specifying some of the contract expressions in a Library Management System.

Initial Avail Status	Actors	Request Triggers	Post-Conditions	Final Avail Status
NonEx or Gone		OK Add Request	& Copy object added & Copy loaned count = 0 & Book copies up 1	
Ores or Cres or Fix		OK Restore Request		
Out	or Borrower	OK Manual or OK Auto Return Request	& Borrower loan list down 1	
	CC: Collection Admin		Common Conditions: Copy status start date = today & Copy availabler id = Staff or Borrower id & Successful <u>action</u> msg returned	CC: Avail

Figure 8: Symmetric State Table for **Copy Availability Status** (Part 1 of 2)

Initial Avail Status	Actors	Request Triggers	Post-Conditions	Final Avail Status
	or Borrower	OK Manual or OK Auto Borrow Request	& Loaned copies up 1 & Borrower loan list up 1	Out
or CRes or Fix		OK Open Reserve Request		ORes
or ORes or Fix		OK Closed Reserve Request		CRes
or ORes or CRes		OK Repair Request		Fix
or Out or ORes or CRes or Fix		OK Remove Request	& Book copies down 1	Gone

CC:	CC:		Common Conditions:	
Avail	Collection Admin		Copy status start date = today & Copy unavailabler id = Staff or Borrower id & Successful <u>action</u> msg returned	

Figure 9: Symmetric State Table for **Copy Availability Status** (Part 2 of 2)

Other contract expressions not involving states, but for the same Library Management System, are specified in vertical tables as shown in the following example.

List Requests	Constant Conditions	PostConditions
List copies borrowed by another OK		Copies borrowed by another list display returned
List overdues OK		Overdue copies list display returned
	Common Conditions: request submitted & request OK	

Figure 10: Vertical Action Contract for **List Request Reactions** (Part 1 of 2)

List Requests	Constant Conditions	PostConditions
List copies borrowed by another NG		
List overdue copies NG		
	Common Conditions: request submitted & request NG	Common Conditions: Invalid <u>list request</u> msg returned

Figure 11: Vertical Action Contract for **List Request Reactions** (Part 2 of 2)

11 Using Contracts

Action contracts can be used to document functionality whenever there is concern about misunderstanding required or existing behavior. Usage can range from one or two expressions to clarify critical ambiguities to the development of a complete contract.

Contracting can occur:

1. prior to acquisition, custom development, or predefined testing,
2. during exploratory testing [Ba01] of unfamiliar, under-documented systems or
3. during the reengineering of under-documented systems.

In most cases, action contracts should be developed incrementally in a series of short sessions with the client to review the newest contract increment as well as identify the next set of issues to address. This is followed by a session of analyst contracting to capture the new issues and then a return to the client for another round of contract increment review and issue identification. While a draft scope statement should exist from the beginning, expressions, or classes, or both may be added and reviewed during any iteration.

Even if functionality information is coming from exploration of an existing system, the contracting should still be incremental. This just substitutes “existing system” for “client” in the description above.

In addition, we can model usage by supplementing contracts with use cases. Precise Usage Models [Ge02] feature contracts supplemented with Precise Use Cases [Ge01].

Contracts can also be supplemented with nonfunctional needs (e.g., security, reliability, or performance), design constraints, and external interface criteria to create a standard requirements specification, e.g., IEEE 830 [Da93].

12 Previous Work

Bertrand Meyer [Me01, Me92a] proposed using pre and post conditions as a form of contract between a user and a developer at the single component or class level of specification. In this framework, the user is responsible for satisfying the pre-condition and the developer is responsible for delivering software that satisfies the post-condition. Software behavior outside the scope of the pre-condition is unconstrained. These ideas have been implemented as contract assertions in the Eiffel language [Me92b] among others [MM01].

Constant conditions in this Design by Contract™ framework refer to properties that hold for all instances of a class, rather than properties that are involved in but unchanged by a specific method of the class. All constant conditions of a class are constant for every method of the class, but they may not be relevant for a specific method. However, other properties that are not constant conditions for the entire class such as the continued existence of a specific object upon access to that object are constant conditions for the access operation.

Others have observed the semantic deficiencies in state diagrams. These include the authors of [KG99] and others that they reference in section 4.1 of that paper.

A less formal approach to contracts can be found in [La02].

13 References

- [Ba01] Bach, James **What is Exploratory Testing?** [Available among the articles at www.satisfice.com]
- [Da93] Davis, Alan M. **Software Requirements : Objects, Functions, and States** Prentice Hall 1993
- [Ge01] Gelperin, David **Precise Use Cases** [Available at www.LiveSpecs.com]
- [Ge02] Gelperin, David **A Q&A Intro to Precise Usage Modeling** [Available at www.LiveSpecs.com]

- [KG99] Kent, S. and Gil, J. **Visualizing Action Contracts in Object-Oriented Modeling**
[Available among the publications at www.cs.york.ac.uk/puml]
- [La02] Larman, Craig **Applying UML and Patterns** Prentice Hall PTR 2002
- [Me01] Meyer, Bertrand **Building bug-free O-O software: An introduction to Design by Contract** [Available at <http://eiffel.com/doc/manuals/technology/contract/index.html>]
- [Me92a] Meyer, Bertrand **Applying Design by Contract** IEEE Computer Vol. 25
No 10 October 1992 pp. 40-51
- [Me92b] Meyer, Bertrand **Eiffel: The Language** Prentice Hall, 1992
- [MM01] McKim, Jim and Mitchell, Richard **Design by Contract by example**
Addison Wesley To be published
- [MO95] Martin, J. and Odell, J. **Object-Oriented Methods** Prentice Hall 1995
- [SM89] Shlaer, Sally and Mellor, Stephan J. **Object-Oriented Systems Analysis: Modeling the World in Data** Yourdon Press 1989
- [WK99] Warmer, Jos and Kleppe, Anneke **The Object Constraint Language: Precise Modeling with UML** Addison-Wesley 1999