A semantics of compensations for long-running transactions.

Tony Hoare. Draft: June 20, 2003.

Abstract

This note provides an executable semantics for an idealised and simplified version of the compensation capability of the draft standard for the Business Process Workflow Language BPEL4WS. It illustrates the use of a semantic technique that might be applicable to a more realistic description of the current draft standard. The ultimate purpose of such a description would be: (1) to assist the standardisation committee in exploration of language design options; (2) to define the required level of consistency between different implementations of the language on different platforms; (3) to permit proof of the correctness and other properties of implementations of the language itself; and (4) to provide a secure foundation for the design of a suite of platform-independent program analysis tools to check the correctness of user programs expressed in the language. The immediate contribution of semantics to any one of these individual goals is expected to be small, but it may be significant cumulatively in the long term.

Introduction

Our executable semantics attempts to deal independently with each major programming language concept. The concept is presented as if it were a design pattern which enables the concept to be exploited in the design of programs expressed in a programming language which does not include it. We will take an unusually rigorous approach to the specification of a design pattern, which in our view consists of the following:

(0) *purpose* : a clear description of the purpose of the concept that is to be implemented. This may be formalised in terms of assertions (preconditions and postconditions) that are true before and after each program construct. (1) *data* : the declaration of additional global or local program variables, which are accessible only by primitive actions of the pattern. (2) *actions* : code for a collection of primitive actions which access and update these variables, and which can be included in the user program. (3) *protocols* : specification of protocols and other conventions (healthiness conditions) that must be observed by the user program to ensure that the pattern achieves its intended purpose. (4) *transformation* : an automatic analysis or translation algorithm, ensuring that the use of the design pattern conforms to the rules given above. (5) *algebra* : valid algebraic laws and correctness principles that may be used by program analysis tools in checking programs that use the pattern.

Our sole purpose in defining a design pattern is to convey a platform-independent understanding of the meaning of a language feature, and how to use it safely and correctly in each particular application. Although the pattern permits direct execution of the program that uses it, there is no need for the execution to be efficient. Furthermore, since execution is only a simulation, it does not have to reflect the architecture of the target execution platform. In fact, our semantic patterns will be executable sequentially on a single machine, even when the program is intended for concurrent or distributed execution on multiple machines. For purposes of efficient

execution in the real world, the real implementation of a programming concept will use features of the platform architecture which are outside the scope of a more abstract semantics. This abstraction from implementation technology is what makes the semantics simpler than the actual implementation, and more useful for its full range of intended purposes.

As a tutorial example of our semantic technique, in the next section we shall explain the semantics of exit-jumps (or returns) by showing how to implement them in a purely structured language that does not have any form of control break. The purpose of the exit is to avoid further processing when it is known that the objective of a program structure has already been achieved. The relevant design pattern introduces a hidden tag variable, which will indicate whether termination of the scope is the normal kind (by reaching the end), or whether it has been accelerated by an explicit exit. The tag variable must be tested accessed and assigned in certain specific ways that conform to healthiness conditions laid down by the design pattern. Of course, in an actual implementation, it is far more efficient to use a jump instruction provided by the architecture of the executing machine.

Our design patterns do not impose restrictions on the choice of language in which they are implemented. We assume only that the language contains assignments, assertions, sequential composition, conditionals, while loops, and declarations. The behaviour of each fragment of program can be understood as a relation between a state of the world before starting its execution, and its state after the fragment has terminated, or reached some other stable state. The state of the world is modelled as an allocation of values to all the global variables of the program and other global variables representing the state of the program environment and the history of interactions between the environment and the program. We will reason quite informally about the programs in this language, using simple operational intuition about the effect of executing the programs. For a formalisation of the base language and for the rules of reasoning about it see [Laws of Programming] or [Unifying Theories of Programming] or [a Discipline of Programming].

The main unfamiliar feature that we will need in the base language is non-determinism: (P or Q) is a program that behaves either like P or like Q; but we neither know nor care which one is actually executed. In the relational interpretation of programs, non-determinism is simply defined as relational union. This interpretation permits a simple definition of program refinement, namely relational inclusion; this states that a more refined program is simply one that is more deterministic. The more refined program is always better, because a deterministic program is easier to predict and control, no matter what purpose we want it for.

The style of this note is tutorial, and assumes only a passing acquaintance with operational semantics, with the relational calculus, and with the concept of a long-running transaction and its compensation.

Tutorial: the exit.

Let us start with a structured procedural programming language, which has no jumps, exceptions or any other form of control break. In this section, we will introduce a design pattern that implements a simple exit facility. The 'exit' statement causes an

immediate jump to the end of some enclosing structure Q, which has been specified as an exit-context by the syntax 'exit_scope(Q)'. Scopes may be nested, and the exit terminates the smallest enclosing exit_scope.

(0) *purpose* : An exit is invoked when the program discovers that it has already achieved the objective of its whole scope, so there is no point in executing the remaining actions of its exit_scope. This objective of the exit-scope is normally specified by a postcondition assert (R); this should automatically be taken as the precondition of any exit statement occurring in this scope (and not, of course, in a more deeply nested one). The program immediately following the exit (if any) never gains control, so the postcondition of any exit is arbitrary. This is indicated by writing 'assert(false)' as the postcondition.

exit_scope(.... ; assert(R) ; exit ; assert(false) ; ...) ; assert(R) ;

(1) *data* : We introduce a hidden global variable into the state of the world. Call it 'tag'. Normally it contains a value called 'normal'; but immediately after an exit it contains a special value 'exit_break'.

(2) *actions* : The hidden variable obtains this value only as a result of assignment by the exit command:

exit is defined as {if tag == normal then tag := exit_break;}

The only place where the value of the tag returns to normal is at the end of an exit scope.

exit_scope(Q) is defined as

[Q ; if tag == exit_break then tag := normal]

(3) *protocol* : We now need to ensure that all normal program statements are ignored when the tag has the exit_break value. One way to do this is to surround each normal statement Q by a conditional that tests the tag, and omits Q unless the tag is normal:

if tag == normal then Q.

In fact, we will use this construction so often, that we introduce an abbreviation

[Q]	is defined as	if tag == normal then Q ;
Q is defined to be normal	if and only if	Q == [Q]

From the first definition, we can prove that double brackets are the same as single brackets, as shown in the theorem:

[[Q]] == [Q]

Proof: expand the definition of [ ] twice, and use the general law of programming that

if b then {if b then Q ;} == if b then Q

From the definition of normality it follows that do_nothing and exit are normal; and from the theorem it follows that [Q] is normal, even if Q is not. The following theorem shows that the normality condition achieves its goal of omitting all code after an exit:

exit ; P == exit if P is normal.

(4) *translation* : An easy way to enforce the protocol of normality is to translate every structure Q of the user program into [Q], on the assumption that Q itself does not mention the tag variable. The gross inefficiency that results can be mitigated by application of the algebraic laws given under (5). In principle, every statement Q of the user program must be surrounded by the brackets [and] , or rather by their defined meaning. So a user program

N; {if c then P else Q }; R

will get translated to

[[N] ; [if c then [P] else [Q]] ; [R]]

After the optimisation in accordance with the rules described below, this may be reduced to

[N ; [if c then { P ; [R] ;} else {Q ; [R] ;}]]

If N (or P or Q) contains no exit, the next following square brackets can be omitted. Also omitted is any material following an exit.

(5) *algebra* : If Q contains an exit, all the statements that follow the exit will be omitted (because they are all normal), until the end of the smallest enclosing exit_scope. Then the tag is set back to normal, to enable the rest of the program to be executed. The working of the construction is illustrated by the following algebraic laws

exit_scope(exit) == do_nothing
exit_scope(do_nothing) == do_nothing
exit_scope(P;Q) == P; exit_scope(Q), if P does not contain an exit

In fact, any exit that occurs as the last action of an exit_scope (e.g., at the end of a limb of a conditional) can be omitted.

Other laws that may be used in optimisation are :

[[P] ; [Q]] == [P ; [Q]]
[[P ; [Q]] ; [R]] == [P ; [Q ; [R]]]
[exit ; [R]] == exit
[P ; [Q]] == [P ; Q], if P contains no exit
[while b do [P]] == while (b) && (tag == normal) do P
[if b then [P] else [Q] ; [R]] == [if b then {P ; [R] ;} else {Q ; [R] ;}]

Exceptions.

The description given so far applies only to the simplest kind of control break. However, it is easily extended to allow more general exception handling. Just regard the exception name as a possible value for the tag. This is tested as a condition for entry to the handler.

(0) *purpose* : An exception is thrown when the program discovers that the objective of the exception_scope can more effectively be achieved in some way other than executing the remaining actions of the scope. The alternative method is provided by the exception handler which is declared with the exception_scope

Exceptions are often invoked when the preferred goal of the program is found to impossible. That is why exceptions are commonly identified with failure rather than success. Our view is more precisely accurate: in principle, the specification of the original task includes an 'if possible' clause, which defines more or less precisely the consequences of invoking the program in circumstances in which its task is impossible. The discovery of this impossibility therefore makes the original specification easier to fulfil. As in real life, impossibility is the perfect excuse.

(1) *data* : no new data is required.

(2) *actions* :

throw(e) is defined as [tag := e]

exception_scope(P, e, Q)

is defined as [P ; if tag == e then {tag := normal ; Q ;}]

The protocols, the translation, and the algebra of exceptions are the same as for exits. A few extra laws for exception_scopes are needed :

exception_scope(throw(e) , e, Q) == Q
exception_scope(do_nothing , e, Q) == do_nothing
exception_scope({P ; R} , e, Q) == P ; exit_scope(R, e, Q)
if P contains no throw(e)

Compensations

In this section we give a similar semantic treatment to the compensation capability modelled on that of the language BPEL4WS. A compensation allows the effect of a program to be undone, or at least partially undone. It is therefore useful for recovery from failure to meet a specification, in those cases where the specification itself does not permit an alternative outcome.

(0) *purpose* : A compensation is a fragment of program that is capable of undoing the external effect of its entire compensation_scope. It can be called only in (some approximation of) the final state that results after its compensation_scope has

terminated normally. The compensation attempts to restore the state of the world as closely as possible to (some approximation of) the state that it had when the compensation_scope first started.

To formalisation of this concept introduces an ordering relation APPROX between states of the world. The predicate s APPROX t means that the state s is an acceptable approximation of an initial state t . Now C is a correct compensation for a scope P if on normal termination, it leaves a final state which is an acceptable approximation of the initial state

{P ; C ;} is a refinement of {APPROX ; assert(tag == normal);}

The meaning of the relation APPROX is entirely application-dependent (in the ideal, it may even be the relationship of equality); and so the proof of the correctness of a compensation is entirely the responsibility of the application programmer. However, the implementation of a compensation capability must preserve correctness when compensation_scopes are assembled into long-running transactions.

(1) *data* : The design pattern for compensation introduces another hidden variable called comp, whose value at all times is itself a program. To invoke the current value of comp will undo the entire effect of the program, back to the point at which this variable was declared and initialised.

(2) *actions* : Obviously, at the moment that the whole program starts, nothing needs to be done to restore the state

initialise_comp is defined as [comp := {do_nothing}]

A long-running transaction is one whose execution consists of a series of atomic (ACID) transactions. At the end of each atomic action P, an interaction with the outside world takes place. This interaction is a commitment because it changes the state of the world, and it cannot be undone by check-pointing or any other automatic technique. Instead, the user has to provide a compensating action C, which is saved up in the comp variable, in case it is needed as a result of a failure that occurs some time later in the program. The compensating action is provided by a statement

addcomp(C) defined as [comp := {C;comp}]

Theorem addcomp(do_nothing) == do_nothing
addcomp(C) ; addcomp(D) == addcomp(D;C)

The last law show the familiar rule that compensations will be executed in reverse order to the actions which they compensate.

When storing the value of a program C, we assume that any declared non-local variables that are mentioned in C are replaced by constants denoting the current values of those variables at the time of the storage. This means that it is safe to call the compensation even after exit from the scope in which its global variables were declared.

It is a basic rule of life that a compensation can be invoked at most once for each time the program that invokes it is executed. This rule is enforced by re-setting the comp variable immediately after it is called, with a value which indicates a programming error if it is ever called again. At the end of the compensation, the tag is set to indicate that termination was not normal

call_compensation is defined as

[new save := comp; comp := {assert(false)} ; call(save) ; tag := compensated]

Theorem call_compensation ; P = call_compensation (because P is normal)
addcomp(C) ; call_compensation == C ; call_compensation
initialise_comp ; call_compensation == initialise_comp

Consider now a long-running transaction P that has successfully performed a series of atomic transactions, and has accumulated a series of compensations in the compensation variable. But perhaps at in the progress of the program has reached a certain stage in which there is now a faster way of compensating for all the actions of P so far, not one at a time in reverse order, but more efficiently in one fell swoop, by executing some alternative compensation D. To achieve this effect, the definition of BPEL compensation scope provides a way of over-riding the effect of all the compensations that have been accumulated during the execution of P, and replacing them by D .

compensation-scope(P, D) is defined as

[new save := {D ; comp ;}; P ; [comp := save]]

Note that the values of the variables of D are frozen at the time of entry into the scope, whereas D itself can never be executed until after P has terminated normally. Also, if

(3) *protocols* : A program P is defined to be correctly compensated if

P ; call_compensation is a refinement of {APPROX ; assert(tag == normal)}

It is the responsibility of the programmer to ensure that each atomic compensation_scope is correctly compensated. It is the responsibility of the designer and implementer of the language to ensure that all the larger structures of the language preserve this property. That responsibility is discharged by proof of the following fundamental theorem of compensation

If P and Q are correctly compensated,
then so are {P;Q;}, if c then P else Q, and while c do P

(4) *translation* : none needed.

(5) *algebra* : The following theorems show that our definitions achieve at least part of their intended effect.

compensation_scope(call_compensation, C) == call_compensation
compensation_scope(do_nothing, C) == addcomp(C)
compensation_scope(compensation_scope(P, C), D) ==
compensation_scope(P, D)

The last theorem shows the over-riding effect of the compensation_scope.

Pick

The compensation capability described so far permits the entire effect of a program to be undone. But we are considering applications in which programs are written to define the behaviour of a system over periods of indefinite length. It is just not acceptable to 'go back to April 1 1983'. The pick capability of BPEL is designed to stop the backtracking execution of compensations at some suitable point, and start moving forward again on some alternative course of action. The construction 'pick_first(P, Q)' first tries P, and succeeds at once if P does so; but if P has failed by calling its own compensation, Q is executed instead of P.

(0) *purpose* : Let P and Q be alternative ways of achieving the same objective R. Suppose it is not reasonable to test in advance which one of them will succeed, but for reasons irrelevant to correctness, P is preferable. Then pickfirst(P, Q) has the same postcondition R as both P and Q. Its precondition is the union of the preconditions of P and Q.

... to be continued.

Semantics in the service of standardisation. Tony Hoare. 7 June 2003

Summary.

This short note summarises the ways in which the formalisation of the semantics of a programming language may assist in its standardisation, implementation and subsequent use. The goals of a formal semantics may be summarised under the following headings.

1. To help a language designer or a standardisation committee to explore design options, particularly for proposed extensions to a draft standard. A semantic definition can reveal and resolve unexpected complexities and interaction effects between a new feature and the main body of the language.

2. To define the required degree of conformity of an implementation to the standard, and so contribute to trouble-free program interchange and inter-operation.

3. To assist in the design of correct, compatible and efficient implementations of the common standard on different platforms.

4. To guide the design of programmer productivity tools, for example program analysers, test harnesses, test case generators, fault injectors, model checkers, certification suites and even program verifiers. These tools should certainly be consistent with the language as standardised and implemented.

5. To assist in the construction of user manuals and educational materials, introducing the minimum number of essential concepts in the right order, with explanation how to use them correctly, efficiently and effectively.

The contribution of a formal semantics to each of these goals individually may be small; but cumulatively and in the longer term, the benefits may be sufficient to justify the effort of formalisation.

One must recognise that formalisation of semantics in the past has made rather rare contributions to programming language design and standardisation. However, the underlying science has made continuing progress, and scale of the problems continues to increase. So we propose to address the question whether this progress is now sufficient to achieve some of the goals listed above. In particular, we will attempt to supply a semantics for certain aspects of the new business choreography language BPEL4WS, now in process of standardisation.

This language is intended for the description of long-running transactions, which engage in communications with other similar transactions, perhaps defined by other businesses, which require their code to be kept secret. Because these transactions are not independent, their failure cannot be compensated by any automatic technique of reverting to a check-pointed state. To undo the effect of the failed transaction on the external world, it is therefore necessary to engage in further communications, retractions, cancellations, apologies, penalty payments, etc. The program that does this is called a compensation; and it has to be written in an application-dependent way

by the original programmer. However, the language does provide structured methods for composing such compensatable transactions into larger and longer-term transactions which are also compensatable. A formal semantics can provide a check of the soundness of the strategy for doing this.

Varieties of semantics.

There are now a number of styles available for the presentation of the semantics of a programming language. Each of them is well suited to describing different aspects of the language, and in achieving various subsets of the goals listed above. Fortunately, it is also known how to use the different styles in combination, and so describe the whole language in a consistent way, with some hope of achieving all the goals simultaneously. To justify this hope would be the objective of the research that we propose.

The main styles of semantic presentation may be summarised under the following headings.

1. A structured operational semantics specifies in an abstract way the individual steps that may be taken by an implementation of the language in executing a program. It is strong in describing interleaving and interaction of concurrently executing components of a system, preferably one that does not update shared memory. The semantics is directly amenable to model checking, which detects the risk of deadlock in concurrent systems. An operational semantics may be presented as an executable PROLOG program; and PROLOG non-determinism may be exploited to explore the full range of non-deterministic possibilities in programs expressed in the original source language.

2. An assertional semantics specifies the proper correspondence between a program and the assertions (preconditions and postconditions) that express what it is supposed to do. Assertional semantics is good for sequential programs that update named internal variables inside the memory of a computer. The assertions are used as oracles in test harnesses, and they can be potentially exploited in test case generation, and even in program verification. Ideally, assertions can help even at the design stage for systems.

3. A denotational semantics translates every program into a mathematical object that exhibits corresponding behaviour in some abstract mathematical domain. It is especially good for programs expressed as higher-order functions, and for applications that make heavy use of recursion, in defining both program structures and data structures. The semantics may be presented as suite of functional programs, expressed in a language like Haskell.

4. An algebraic semantics is a presentation of a more or less complete set of algebraic equations between programs written in the same language. The equations may be used as correctness-preserving transformations in program optimisation, both manual and automatic. Equations are surprisingly good at expressing or even conveying an engineering intuition about the intention and the meaning of a program feature. If sufficient equations are given to support a normal form, they can be executed in an algebraic system like Maude.

5. A translational semantics translates the programs of a complex language into a simpler subset of its own language, which may itself have a semantics of a different kind. The result of the translation may be a fully instrumented program, which can be executed directly as a test harness, or subjected to a more penetrating program analysis to reveal its errors, or even to prove that it has none. The semantics can be presented as a collection of design patterns, representing each concept independently.

Where the semantics of a language is presented in an executable form, the execution does not have to be efficient; nor does it have to exploit the concurrent or distributed structure of the target architecture on which the real programs will be run. Abstraction from the realistic concerns of a practical implementation is the only reason for hoping that a formal semantics will be any simpler than the implementation itself. And if it is not significantly simpler, the semantics will fail in its original goal of aiding the understanding of its users, its implementers, and its designers.

Modularity.

One of the most important aids to understanding in the presentation of semantics is its modularity. Modularity can be achieved in three orthogonal ways.

1. Each feature of the language should be defined independently of all other features. It should be possible to prove properties of each feature directly from its definition. Of course, this is only possible for a language which has been designed to be reasonably free of feature interaction.

2. Each module of a program should have a meaning which is understandable independently of the context in which it is run. Thus each module can be tested, analysed, and even verified independently of the much larger system in which it is embedded.

3 Wherever possible, the features and control structures of the language should be defined in terms of binary operators that are associative. Associativity ensures that a structure consisting of a long string of operands can be understood by considering only one pair of adjacent operands at a time.

We propose to explore a combination of these semantic techniques in application to some of the features of BPEL4WS. We will start with a simple language due primarily to Dijkstra, which has a clearly established assertional semantics. It is already effectively a subset of all procedural languages, including BPEL. We will then use the translational technique to define other more advanced and specialised features of the language. The translation will produce an executable test harness, fully instrumented by the relevant assertions. The relevant algebraic laws will be derived wherever possible as theorems.

We would then like to present a denotational semantics, by implementing the same collection of features in the higher order functional language Haskell. Monads are a

splendid way to maintain the desirable modularity. Finally, it would be interesting to explore PROLOG as a directly executable presentation of an operational semantics.