Compensable transactions

Tony Hoare
Microsoft Research, Cambridge, England

Summary. The concept of a compensable transaction has been embodied in modern business workflow languages like BPEL. This article uses the concept of a box-structured Petri net to formalise the definition of a compensable transaction. The standard definitions of structured program connectives are extended to construct longer-running transactions out of shorter fine-grain ones. Floyd-type assertions on the arcs of the net specify the intended properties of the transaction and of its component programs. The correctness of the whole transaction can therefore be proved by simple local reasoning.

1. Introduction.

A compensable transaction can be formed from a pair of programs: one that performs an action and another that performs a compensation for that action if and when required. The forward action is a conventional atomic transaction: it may fail before completion, but before failure it guarantees to restore (an acceptable approximation of) the initial state of the machine, and of the relevant parts of the real world. A compensable transaction has an additional property: after successful completion of the forward action, a failure of the next following transaction may trigger a call of the compensation, which will undo the effects of the forward action, as far as possible. Thus the longer transaction (this one together with the next one) is atomic, in the sense that it never stops half way through, and that its failure is adequately equivalent to doing nothing. In the (hopefully rare) case that a transaction can neither succeed nor restore its initial conditions, an explicit exception must be thrown.

The availability of a suitable compensation gives freedom to the forward action to exercise an effect on the real world, in the expectation that the compensation can effectively undo it later, if necessary. For example, a compensation may issue apologies, cancel reservations, make penalty payments, etc. Thus compensable transactions do not have to be independent (in the sense of ACID); and their durability is obviously conditional on the non-occurrence of the compensation, which undoes them. Because all our transactions are compensable, in this article we will often omit the qualification.

We will define a number of ways of composing transactions into larger structures, which are also compensable transactions. Transaction declarations can even be nested. This enables the concept of a transaction to be re-used at many levels of granularity, ranging perhaps from a few microseconds to several months -- twelve orders of magnitude. Of course, transactions will only be useful if failure is rare, and the longer transactions must have much rarer failures.

The main composition method for a long-running transaction is sequential composition of an ordered sequence of shorter transactions. Any action of the sequence may fail, and this triggers the compensations of the previously completed transactions, executed in the reverse order of finishing. A sequential transaction succeeds only if and when all its component transactions have succeeded.

In the second mode of composition, the transactions in a sequence are treated as alternatives: they are tried one after another until the first one succeeds. Failure of any action of the sequence triggers the forward action of the next transaction in the sequence. The sequence fails only if and when all its component transactions have failed.

In some cases (hopefully even rarer than failure), a transaction reaches a state in which it can neither succeed nor fail back to an acceptable approximation of its original starting state. The only recourse is to throw an exception. A catch clause is provided to field the exception, and attempt to rectify the situation.

The last composition method defined in this article introduces concurrent execution both of the forward actions and of the backward actions. Completion depends on completion of all the concurrent components. They can all succeed, or they can all fail; any other combination leads to a throw.

2. The Petri box model of execution.

A compensable transaction is a program fragment with several entry points and several exits. It is therefore conveniently modelled as a conventional program flowchart, or more generally as a Petri net. A flowchart for an ordinary sequential program is a directed graph: its nodes contain programmed actions (assignments, tests, input, output, ... as in your favourite language), and its arrows allow passage of a single control token through the network from the node at its tail to the node at its head.. We imagine that the token carries with it a value consisting of the entire state of the computer, together with the state of that part of the world with which the computer interacts. The value of the token is updated by execution of the program held at each node that it passes through. For a sequential program, there is always exactly one token in the whole net, so there is never any possibility that two tokens may arrive at an action before it is complete.

In section 6, we introduce concurrency by means of a Petri net transition, which splits the token into separate tokens, one for each component thread. It may be regarded as carrying that part of the machine resources which is owned by the thread, and communication channels with those parts of the real world for which it is responsible. The split token is merged again by another transition when all the threads are complete. The restriction to a single token therefore applies within each thread.

A structured flowchart is one in which some of its parts are enclosed in boxes. The fragment of a flowchart inside a box is called a block. The perimeter of a box represents an abstraction of the block that it contains. Arrows crossing the perimeter are either entries or exits from the box. We require the boxes to be either disjoint or properly nested within each other. That is why we call it a structured flowchart, though we relax the common restriction that each box has only one entry and one exit arrow. The boxes are used only as a conceptual aid in planning and programming a transaction, and in defining a calculus for proving their correctness. In the actual execution of the transaction, they are completely ignored.

We will give conventional names to the entry points and exit points of the arrows crossing the perimeter of the box. The names will be used to specify how blocks are composed into larger blocks by connecting the exits of one box to the entries of another, and enclosing the result in yet another box. This clearly preserves the disjointness constraint for a box-structured net.

One of the arrows entering the box will be designated as the *start* arrow. That is where the token first enters the box. The execution of the block is modelled by the movement of the token along the internal arrows between the nodes of the graph that are inside the box. The token then can leave the box by one of its exit points, generally chosen by the program inside the box. The token can then re-enter the box again through one of the other entry points that it is ready to accept it. The pattern of entering and leaving the block may be repeated many times.

In our formal definition of a compensable transaction, we will include a behavioural constraint, specifying more or less precisely the order in which entry and exit points can be activated. The behavioural constraint will often be expressed as a regular expression, whose language defines all permissible sequences of entry and exit events which may be observed and sequentially recorded.

We will introduce non-determinism into our flowchart by means of the Petri net place. A place is drawn as a small circle (Figure 2.1) with no associated action. It may have many incoming arrows and many outgoing arrows. The place is entered by a token arriving along any one of its entries. The next action (known as a firing) of the place is to emit the token just once, along any one of its exit arrows. The token arriving at the exit of the place may have originated at any one of its entries. The strict alternation of entries and exits of a place may be formally described by the regular expression

$$(l + m + n) ; (r + s + t)$$

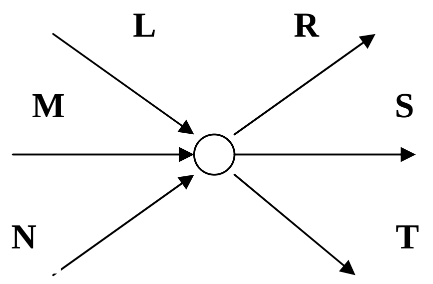
where ***l, m, n*** name the entries of the place, and ***r, s, t*** name the exits.

Technical note: in general, a Petri net place is capable of storing a token. In our restricted calculus this capability is exploited only once (in section 6). In fact, we may regard a token as passing instantaneously (as a single event) through any sequence of consecutive states. Of course, a regular expression cannot model this simultaneity.

If the place has only a single exit arrow, it acts as a normal fan-in, and has the same function as in a conventional flowchart. If there are many exits, the place acts as a fan-out. The choice of exit arrow on each occasion of entry is usually determined by the current readiness of the block at the head of the exit arrow to accept entry of the token. But if more than one block is ready, the choice is non-deterministic, in the usual sense of don't-care or demonic non-determinism. It is the programmer's responsibility to ensure that all choices are correct; and the implementation may choose any alternative according to any criterion whatsoever, because it is known that correctness will not be affected. For example, efficiency and responsiveness are among the more desirable of the permissible criteria.

We follow Floyd's suggestion that the arrows in a flowchart should be annotated with assertions. Assertions are descriptions of the state of the world (including the state of the machine), and the programmer intends that they should be true of the world value carried by the control token, whenever it passes along the annotated arrow. An assertion on an arrow which enters a box serves as a precondition for the block, and it is the responsibility of the surrounding flowchart to make it true before transmitting the token to that entry. An assertion on an exit arrow from the box serves as a postcondition, and it is the responsibility of the block itself to make it true before transmitting the token through that exit.

The correctness of the composed flowchart may be determined locally in the usual way, by considering the assertions on each arrow and on each place. For an arrow which connects a single exit point to a single entry (usually on another box), the exit assertion of the box at the tail of the arrow must logically imply the entry assertion of the box at its head. For a place, the rule is a natural extension of this. A place is correct if the assertion on *each* one of its entry arrows logically implies *every* one of the assertions at the heads of its exit arrows. In other words, the verification condition for a place is that the disjunction of all the tail assertions implies the conjunction of all the head assertions (see Figure 2.1, where the upper case letters stand for the arrows annotated by the corresponding lower case letter). Thus overall correctness of the entire flowchart can be proved in a modular fashion, just one arrow or place or action at a time.

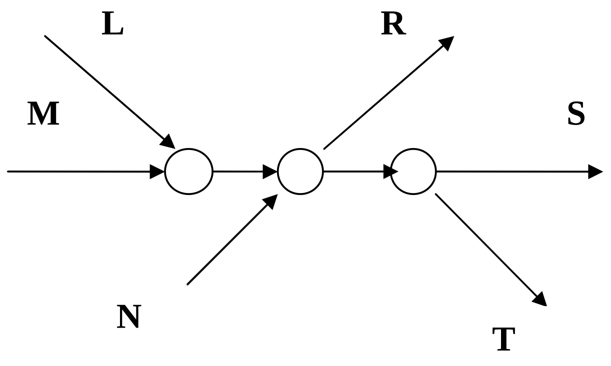


Correctness condition: $\mathbf{L \vee M \vee N \Rightarrow R \wedge S \wedge T}$
Behaviour: $\boldsymbol{(l + m + n) ; (r + s + t)}$

Figure 2.1. A Petri net place

The intention of drawing a box of a structured flowchart is that the details of the flowchart inside the box should be irrelevant to the rest of the flowchart that lies outside the box. From the outside, you only need to know three items: (1) the names of the entry and exit points; these are used to specify how the box is connected into its environment (2) the assertions on the arrows that enter and leave the box; and (3) the constraints that govern the order of entry and exit events, by which the token enters and leaves the box along the arrows that cross the perimeter. If two boxes have the same assertions and the same set of behaviours, we define them to be semantically equivalent.

This rule of equivalence may be applied to just a single place. As a result, any complete collection of linked Petri net places, in which all the entries are connected to all the exits, can be replaced by a single place, -- one that has all the same entries and the exits, but the internal arrows areeliminated. Figure 2.2 therefore has the same semantics as Figure 2.1.



Correctness condition: $\mathbf{L} \vee \mathbf{M} \vee \mathbf{N} \Rightarrow \mathbf{R} \wedge \mathbf{S} \wedge \mathbf{T}$
Behaviour: $\boldsymbol{(l + m + n) ; (r + s + t)}$

Figure 2.2. The same place as Figure 2.1.

3. Definition of a transaction.

A compensable transaction is a special kind of box in a Petri net. It is defined as a box whose names, behaviour and assertions satisfy a given set of constraints. The first constraint is a naming constraint.
A transaction box has two entry points named *start* and *failback*, and three exit points named *finish, fail* and *throw*. The intended function of each of these points is indicated by its name, and will be more precisely described by the other constraints. When a transaction is represented as a box, we introduce the convention that these entries and exits should be distributed around the perimeter as shown in Figure 3.1. As a result, our

diagrams will usually omit the names, since the identity of each arrow is indicated by its relative position on the perimeter of the box.

A more significant part of the formal definition of a transaction is a behavioural constraint, constraining the order in which the token is allowed to enter and exit the block at each entry and exit point. The constraint is conveniently defined by a regular expression:

> *start ; (finish ; failback)* ; (fail + throw + finish)*

This expression stipulates that the first activation of the transaction is triggered by entry of the token at the *start* point. The last de-activation of the transaction is when the token leaves at any one of the three exit points. In between these two events, the transaction may engage in any number of intermediate exits and entries. On each iteration, it finishes successfully, but is later required to compensate by a *failback* entry, triggered by failure of the following sequentially composed transaction. The number of occurrences of *finish* followed by *failback* is not limited, and may even be zero. Typical complete behaviours of a transaction are:

> *start, finish*
> *start, finish, failback, fail*
> *start, finish, failback, finish*
> *start, finish, failback, finish, failback, throw*

The final constraint in the definition of a transaction governs the assertions on its entry and exit points. This constraint expresses the primary and most essential property of a transaction: that if it fails, it has already returned the world to a state that is sufficiently close to the original initial state.

Sufficient closeness might be defined in many ways, but we give the weakest reasonable definition. Our simple requirement is that on failure the world has been returned to a state which again satisfies the initial precondition of the transaction. More precisely, the assertion on the *fail* exit, must be the same original precondition that labels the *start* entry point. Similarly, on *failback* the transaction may assume that the postcondition that it previously established on finishing is again valid. These standard assertional constraints are indicated by the annotations in Figure 3.1. There is no constraint on the assertion ***E*** labelling the *throw* exit.

Many of the constructions of our calculus of transactions can be applied to transactions which satisfy weaker or stronger assertional constraints than the standard described above. For example, a transaction may be *exactly* compensable if on failure it returns to exactly the original state (where obviously the state of the world must be defined to exclude such observations as the real time clock). A weaker constraint is that the postcondition of failure is merely implied by the precondition. Finally, there is the possibility that the transaction has no assertional constraint at all. We will not further consider these variations.

In drawing diagrams with many boxes, the assertions on the arrows will often be omitted. It is assumed that in concrete examples they will be restored in any way that satisfies the intended assertional constraint, and also satisfies the local correctness criterion for assertions, which apply to all arrows and all places.

That concludes our semantic definition of the concept of a transaction. The flowchart gives an operational semantics, describing how the transactions are executed. The assertions give an axiomatic semantics, describing how the transactions are specified and proved correct. The interpretation of a flowchart makes it fairly obvious that the operational and the axiomatic definitions are in close accord.

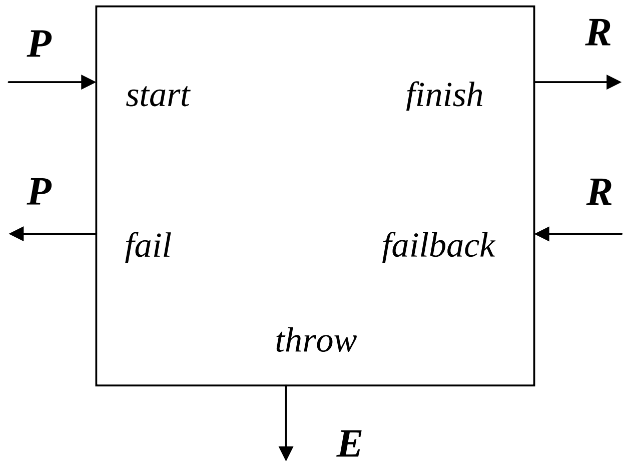


Figure 3.1. Entry and exit names

4. A Calculus of Transactions.

In this section we will define a small calculus for design and implementation of transactions. They are built up by applying the operators that we define to smaller by building them from smaller component transactions. The ultimate components are ordinary fragments of sequential program. Our semantic definitions will mainly use the pictorial representation shown in Figure 3.1. But for the sake of completeness, here is a more conventional syntax.

<transaction> ::= <composed transaction> | *<primitive transaction>*
<primitive transaction> ::= ***succeed*** | ***fail*** | ***throw*** **|**
                                                  <*transaction declaration>*
<transaction declaration> ::= [*<forward action>* ***comp*** *compensation>*]
<forward action> ::= <ordinary program>
<compensation> ::= <ordinary program>
<composed transaction> ::= <sequential composition>
                  | *<alternative composition>* | *<exception block>* |
                  <non-deterministic choice>
<sequential composition>::= <transaction> ***;*** *<transaction>*
<alternative composition> ::= <transaction> ***else*** *<transaction>*
<exception block> ::= <transaction> ***catch*** *< transaction>*
<non-deterministic choice> ::= <transaction> ***or*** *< transaction>*

The shortest primitive transactions are those that do nothing. There are three ways of doing nothing: by succeeding, by failing, or by throwing. Definitions for these transactions are given diagrammatically in Figure 4.1, where the small unconnected arrows will never be activated. The leftmost example does nothing but succeed, and this can obviously be compensated by doing nothing again. The other two examples do nothing but fail or throw. These will never be called upon to compensate.

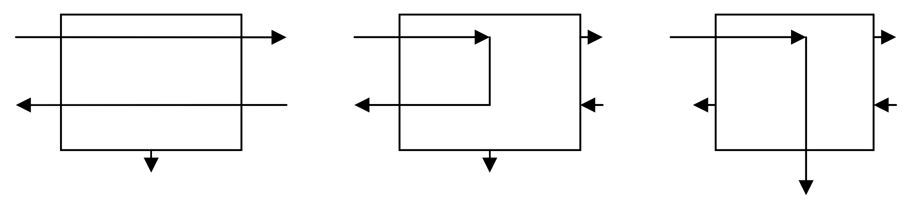


Figure 4.1. Primitive Transactions

Longer running transactions are constructed by composing smaller ones in sequence, or as alternatives, or as a non-deterministic choice, or by try/catch clauses. In all cases, the result of the composition of compensable transactions will also be a compensable transaction. The semantics of each construction will be explained diagrammatically as a box that encloses the boxes representing the components. Some of the exit arrows of each component box will be connected to some of the entry arrows of the other component box, and thereby become internal arrows that can be ignored from outside. Entries and exits on the surrounding box are connected to remaining exits and entries of the component boxes, often ones that share the same name. Where necessary, places may be introduced to deal with fan-in and fan-out.

The basic primitive fine-grained transaction is declared by specifying two sections of normal sequential code (Figure 4.2). The first of them ***T*** performs the required action as control passes from the *start* on the left to the *finish* on the right. The second section of code ***U*** specifies how the action should be compensated as control passes back from the *failback* on the right to the *fail* on the left. Either the action or the compensation can throw, on detecting that neither progress nor compensation is possible. The fan-in of the *throw* arrow indicates that it is not known from the outside which of the two components has actually performed the throw. This fan-in of throws is common to most of the operators defined below, and will sometimes be omitted from the diagrams

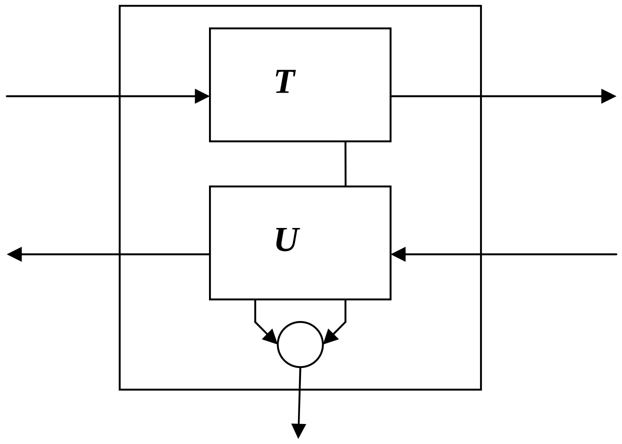


Figure 4.2. Transaction Declaration : [*T comp U*]

The first definition of the constructions for non-primitive transactions will be sequential composition, which is shown in Figure 4.3. The outer block denoting the whole composition starts with the *start* of the first component block **T**. The *finish* of this block triggers the *start* of the second component block **U**. The *finish* of the second block finishes the whole sequential composition. A similar story can be told of the backward-going failure path, which performs the two compensations in the reverse order to the forward operations. This is what makes the composed transaction compensable in the same way as its components are. Furthermore, the sequential composition will satisfy the behavioural constraint for transactions, simply because its components do so.

There should be assertions on each of the arrows. However, the permitted patterns for these assertions are completely determined by the correctness principle for flowcharts, so there is no need to mention them explicitly.

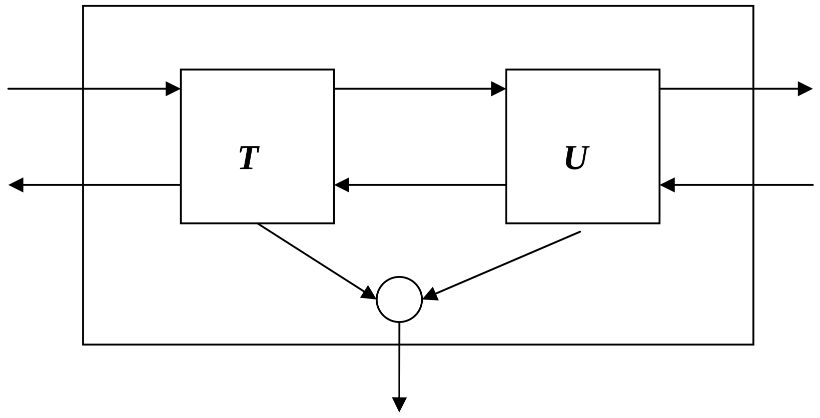


Figure 4.3. Sequential Composition: *T ; U*

This definition of sequential composition is associative and has ***succeed*** as its unit. A simple proof of associativity is obtained by drawing the diagram for a sequential composition with three components, and adding an extra box, either around the two left operands or around the two right operands. It is easy to see that this represents the two different bracketings of the associative law. The flowchart itself remains the same in both cases.

The definition of sequential composition states that failure of any component transaction of the sequence will propagate inexorably to the left, until everything that has ever been done since the beginning of time has been undone. This is not always desirable. The ***else*** operator shown in Figure 4.4 gives a way of halting the stream of failures and compensations. It reverses again the direction of travel of the token, and tries a different way of achieving the same eventual goal.

At most one of these alternatives will actually take effect. The first of them is tried first. If it fails (having compensated of course), the second one is started. If this now succeeds, control is passed to the following transaction, so that it too may try again. As a result, the *finish* exit of the whole composition may be activated twice, or even more often if either of the alternatives itself finishes many times.

Note the fan-in at the *finish* exit: from the outside it is impossible to distinguish which alternative has succeeded on each occasion. Note also the fan-out of the *failback* arrow. In spite of this fan-out, the ***else*** construction is deterministic. When failback occurs, control may pass to the *failback* of either of the alternatives. The selection of destination will always be determined by the behavioural constraint on the component boxes. As a result, control will pass to the alternative that has most recently finished, which is obviously the right one to perform the appropriate compensation.

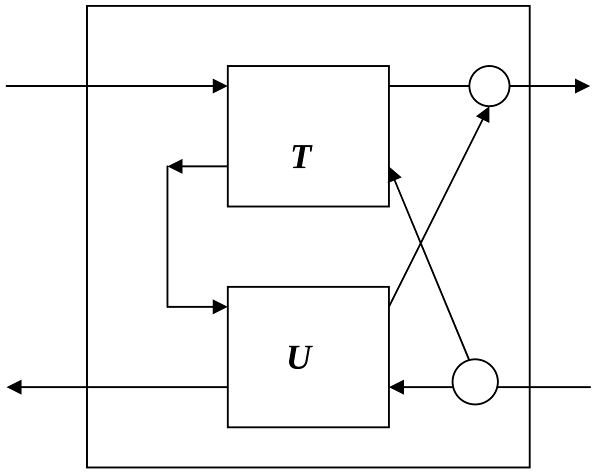


Figure 4.4. Alternative Composition: *T else U*

In the uncertain world in which computers operate, especially in the vicinity of people, it is quite possible that a transaction that has failed once may succeed when it is simply tried again. But clearly the programmer should control how often to repeat the attempt. For example, suppose it is known that the transaction ***U*** is strictly compensable. Then the transaction

succeed else succeed else succeed ; U

merely causes ***U*** to be repeated up to three times – that is, up to three times more often than this transaction itself is repeated by its own predecessors.

The ***else*** operator is associative with unit ***fail*** . The proof is no more difficult than that for sequential composition.

Because of its deterministic order of execution, the ***else*** command is asymmetric. Sometimes the programmer does not care which choice is made, and it is acceptable to delegate the choice to the implementation. For this reason, we introduce an ***or*** constructor, which is symmetric but non-deterministic. Its pictorial definition is very regular, but too cluttered to be worth drawing explicitly. Each entry of the outer box fans out to the like-named entry of both inner boxes. Each exit from the outer box fans in from the like-named exits of the two inner boxes. The non-determinism is introduced by the fan-out of the *start* arrow, which leads to two entries that are both ready to accept the token. After the start, the behavioural constraint ensures that the rejected alternative will never obtain the token. Note that the *fail* arrow of the whole box fans in from the *fail* arrows of both its operands. This shows that the whole box may fail if **either** of its two operands fails. In this respect, non-determinism differs from (and is worse than) the **else** construction, which guarantees to recover from any single failure.

There is yet a third form of choice between alternatives, which plays the role of the external choice in a process algebra. It is denoted by **[]** in CSP or + in CCS. External choice is slightly more deterministic than ***or***, and a bit less deterministic than ***else*** . Like ***or*** it is symmetric. Like ***else*** it recovers from any single failure. It is defined by means of an ***else*** , where the order of trying the operands is non-deterministic.

T [] U = (T else U) or (U else T)

A picture of this operator would have to contain two copies of each of the operands; it is not worth drawing. A conventional equational definition is to be preferred.

This construction ***T [] U***

- fails if both ***U*** and ***T*** fail
- does ***U*** if ***T*** fails
- does ***T*** if ***U*** fails
- chooses non-deterministically if neither fails
- may throw if either ***T*** or ***U*** can do so.

A ***catch*** is similar to an ***else*** in providing an alternative way of achieving the same goal. The difference is that the first operand does not necessarily restore its initial state, and that the second operand is triggered by a *throw* exit instead of a *fail* exit from the first operand. A throw is appropriate when the first operand has been unable either to restore the initial state or to finish successfully. The catching clause is intended to behave like the first operand should have done: either to complete the compensation and fail, or to succeed in the normal way, or else to throw again to some yet more distant catch. Note that the catching clause does not satisfy the assertional constraint for a compensable transaction, because the assertion at its *start* is not the same as the assertion at its *fail* exit.

5. Nested transactions.

We have described in the previous section how a primitive transaction can be declared by specifying a forward action together with its compensation. In the elementary case, both of these are ordinary sequential programs. In this section we will also allow the forward action to be itself a long-running transaction (which we call the child transaction), nested inside a larger parent transaction declaration, as shown in Figure 5.1. As before, the compensation **U** of the parent transaction is an ordinary sequential program, and is triggered from the *failback* entry of the parent transaction. As a result, the *failback* entry of the child transaction **T** is never activated. As a result, when the parent transaction is complete, an implementation can discard the accumulated child compensations, and recover the stack frames and other declared resources of the child transactions.

Nested transactions can be useful as follows. When a long sequence of transactions all succeed, they build up a long sequence of compensations to be executed (in reverse order) in the event of subsequent failure. However, at a certain stage there may be some much better way of achieving the compensation, as it were in a single big step right back to the beginning, rather than in the sequence of small steps accumulated by the child transactions. The new single-step compensation is declared as the compensation for the parent transaction. An example can be taken from a word processing program, where each child transaction deals with a single keystroke, and undoes it when required to compensate. However, when the parent document is complete, any subsequent failure will be compensated by restoring the previous version of the whole document.

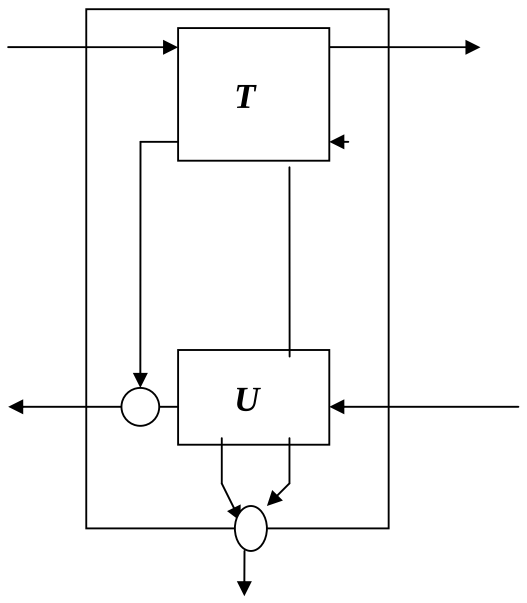


Figure 5.1. Nested Transaction Declaration.

When the child transactions have all finished, their *failback* entries will never subsequently be activated, because (when necessary) the parent compensation is called instead. As a result, at the *finish* of the parent transaction an implementation can simply discard the accumulated child compensations, and recover the stack frames that they occupied.

In addition to stack frames, there may be other resources which need to be released by the child transactions on completion of the parent transaction. In the case of failure, the compensation can do this. But if all the child transactions succeed, we need another mechanism. To provide this requires a significant extension to our definition of a transaction. We add to every transaction (the child transactions as well as the parent) a new entry point called *finally,* placed between the *start* and the *fail*, and a new exit point called *complete,* placed between the *finish* and the *failback.* The nestable transaction declaration therefore takes a third operand, a *completion* action; it is entered by the *finally* entry and exited by the *complete* exit.

When transactions (parents or children) are composed sequentially, their completions are also composed sequentially, like their compensations, by connecting the *complete* exit of the left operand to the *finally* entry of the right operand. So the connecting arrows between completions go from left to right, and the completions are executed in the same order as the forward actions, rather than in the reverse order.

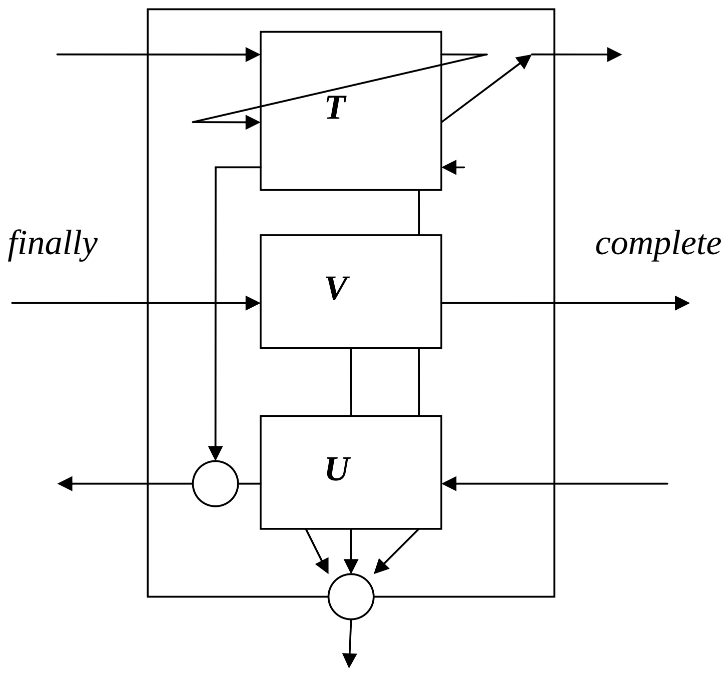


Figure 5.2. Nesting with Completion
[*T finally V comp U*]

In Figure 5.2, the child transaction is denoted ***T***, the parent compensation is ***U***, and the parent completion is ***V***. The child transaction also has a *finally* entry and a *complete* exit, and a completion action, which is not shown explicitly in the diagram. In the case that the child is not a transaction, an ordinary sequential program can be artificially made into a transaction by adding a completion action that does nothing. In that case, the definition of a nested transaction becomes equivalent to that of an un-nested one.

When the whole child transaction has finished, the completions accumulated by the child transactions are triggered. That is indicated in Figure 5.2 by the transverse arrow from the *finally* exit of the child transactions ***T*** to the new *finally* entry of the child transactions themselves. It executes the completion actions of all the children, in the same order as the execution of forward actions of the children.

Another benefit of the introduction of completion actions is to implement the *lazy update* design pattern. Each child makes a generally accessible note of the update that it is responsible for performing, but lazily does not perform the update until all the child transactions of the same parent have successfully finished. On seeing the note, the forward action of each subsequent child takes account of the notes left by all previously executed children, and behaves as if the updates postponed by all previous children had already occurred. But on completion of the parent transaction, the real updates are actually performed by the *completion* code provided by each component child transaction. As a result, the rest of the world will never know how lazy the transaction has been. The completion codes will be executed in the same sequence as the forward actions of the children. Compensations for lazy transactions tend to be rather simple, since all that is required is to throw away the notes on the actions that should have been performed but have not yet been.

Introduction of the new *finally* entry and *complete* exit for completion actions requires an extension to the definition of the behavioural constraint on transactions. Note that a completion is not allowed to fail, though it may still throw.

$$start\ ;\ X$$

where $X \;=\; fail + throw + (finish\ ;\ (\ finally\ ;\ (\ complete + throw\ ) + failback\ ;\ X\ ))$

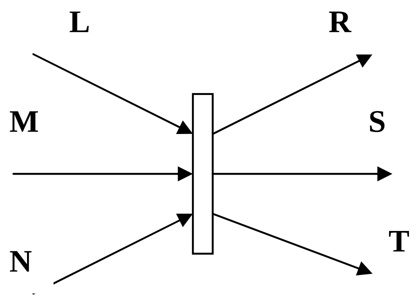
The definition of sequential composition and other operators needs to be adapted to accommodate the addition of new entry and exit points for the completion actions. The adaptations are fairly obvious, and we leave them to the interested reader.

The nesting of transactions may seem unpleasantly complex, but the concept of nesting is essential to deal with the wide range of granularity at which the concept of atomicity can be applied. Many kinds of transaction will last a few microseconds, whereas others may last a few months.

6. Concurrency.

The Petri net place has provided a mechanism for fan-in and fan-out of the arrows of a flowchart. Each activation (firing) of the place involves the entry of a single token along a single **one** of the entry arrow, and the exit of the same token along any **one** of its exit arrows. As a result, a place always maintains the number of tokens in the net – in our treatment so far, there has only been just one token.

Introduction and elimination of tokens from the net is the purpose of the other primitive element of a Petri net, the transition. This too provides a form of fan-in and fan-out, but its behavioural rule is conjunctive rather than disjunctive, universal rather than existential. Each firing of a transition requires the entry of a token on **all** of its entry arrows, and the emission of a token on **all** of its exit arrows. The notation used for transitions is shown in Figure 6.1.



Correctness condition: **L & M & N $\Rightarrow$ R & S & T**
Behaviour: ***(l || m || n) ; (r || s || t)***

Figure 6.1. Petri net transition

If there is only one entry to a transition, it acts as a fan-out: its firing will increase the number of tokens travelling simultaneously in the network. This could certainly lead to confusion if one of the tokens ever meets another at the same place. By allowing only limited and well-structured forms of composition, our calculus will confine each token to a disjoint region of the net, and ensure that tokens meet only at the entry to a transition, which is what is intended. Often, such a meeting place is a fan-in; it has only one exit, so that it reduces the number of tokens in the system.

It is possible to think of all the entry and exit events for a transition as occurring simultaneously. However, in representing this simultaneous behaviour as a regular expression, it is common to use a total ordering of events, in which any causal event occurs before its effect. Furthermore, arbitrary interleaving is commonly used to record sequentially events that occur simultaneously. The regular expression (P || Q) will stand for the set of all interleavings of a string from P with a string from Q . Thus the behavioural constraint on a transition in Figure 6.1 is that the arrival in any order of a token on all of the entry arrows will trigger the emission of a token on each and every one of the exit arrows, again in any order.

The correctness of a transition obviously requires that **all** the assertions on all the exit arrows must be valid at the time of firing. In this respect, the transition is like a place. It differs from a place in the precondition that **all** the assertions on the entry arrows may be assumed to be true when the transition fires. Thus the correctness condition on a transition is that the conjunction of all the entry assertions must logically imply the conjunction of all the exit assertions. In general, there is a possibility that the conjunction will be inconsistent; but we will design our calculus carefully to avoid this risk.

The semantics of the Petri net transition is given in terms of its correctness condition and its behaviour. Thus it satisfies the same equivalence criterion as the place: any acyclic network of pure transitions (in which every external exit is reachable from every external entry) is equivalent to a single transition with exactly the same external entry and exit arrows, but omitting the internal arrows.

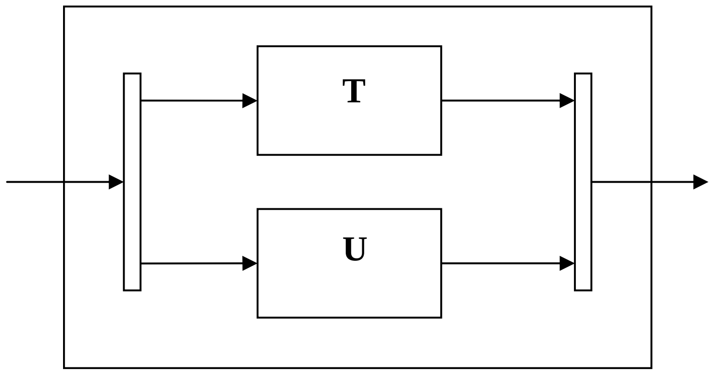


Figure 6.2. Parallel composition: T || U

We will explain the concept of well-structured concurrency first in the context of ordinary programs, which have only a single entry and a single exit arrow. Concurrent composition of two such programs is made to satisfy the same constraint, as shown in Figure 6.2. This shows how two sections of code **T** and **U** will start simultaneously and proceed concurrently until they have both finished. Only then does the concurrent combination finish.

It is evident from the diagram (and from the structured property of boxes) that the only meeting point of the two tokens generated by the fan-out transition on the left will be at the final fan-in transition on the right, where they are merged. The diagram can easily be adapted to deal with three or more threads. But this is not necessary, because the rules of equivalence for transitions ensure that concurrent composition is both an associative and a commutative operator.

The proof of correctness of concurrent threads should be modular in the same way as proof of correctness of all the other forms of composition. In order to make this possible, some disjointness constraints must be placed on the actions of the individual threads. The simplest constraint is that no thread can access any variable updated by some other concurrent thread. This same constraint must also be applied to the assertions used to prove correctness of each thread. The token which travels within a thread can be regarded as carrying the variables and owned by that thread, together with their values.

In simple cases, the constraint on disjointness can be enforced by a compile-time check on the global variables accessed by a thread. But in general, the use of indirect addressing (for example, in an object-oriented program) will make it necessary to prove disjointness by including some notion of ownership into the assertion language. Separation logic provides an elegant and flexible means of expressing disjointness of ownership, and establishing it by means of proof. However, we will not pursue this issue further here.

The disjointness constraint is effective in ensuring consistency of the final assertions of the threads when they all terminate together. It also avoids race conditions at run time, and so prevents any form of unwanted interference between the activities of the threads. However, it also rules out any form of beneficial interaction or cooperation between them. In particular, it rules out any sharing of internal storage or communication channels. A safe relaxation of this restriction is provided by atomic regions (or critical sections). This is defined as a section of code inside a thread, which is allowed to access and update a shared resource. The implementation must guarantee (for example by an exclusion semaphore) that only one thread at a time can be executing inside an atomic region, so race conditions are still avoided. The overall effect of multiple threads updating the shared resource includes an arbitrary interleaving of the execution of their complete atomic regions.

The Petri net formalisation of an atomic region models a shared resource as a token, which may be regarded as carrying the current state and value of the resource. At the beginning of an atomic region, a thread acquires ownership of this token in order to access and update the shared resource; and at the end of the region the shared resource is released. Of course, execution of the region also requires the normal sequential token of the thread that contains it.

An atomic region is defined (Figure 6.3) as a sort of inverse of concurrent composition, with a fan-in at the beginning and a fan-out at the end. For simplicity, we assume that there is only a single shared resource, consisting of everything except the private resources of the currently active individual threads. In most practical applications, many separate resources will need to be declared, but we shall not deal with that complexity here.

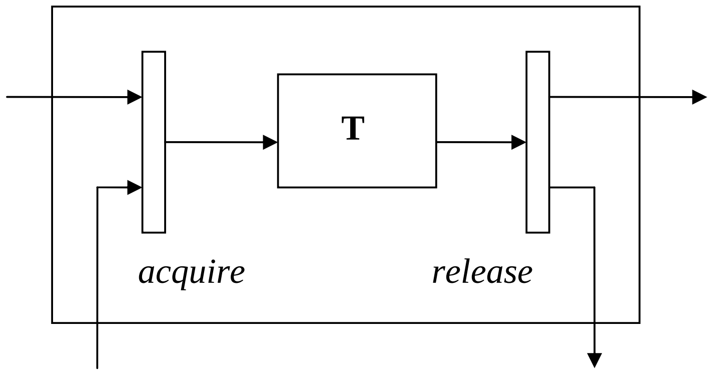


Figure 6.3. Atomic Region: atomic[T]

The definition of an atomic region requires the introduction of another entry and another exit into the standard repertoire. The entry carries the suggestive name *acquire* , and the exit is called *release* . The new entries and exits require extension of the behavioural constraint, by inserting (*acquire;release*)* between every entry and the next following exit. The definition of all the operators of our calculus must also be extended: but this is very simple, because in each diagram defining an operator, all the *acquire* entries are connected via a fan-out place, and all the *release* exits are connected via a fan-in place.

The declaration of a sharable resource is shown in Figure 6.4. The token that represents a resource is created by means of a transition fan-out. The block **T** contains all the multiple threads that are going to share the resource. When all of them have finished, the token is therefore merged again. The assertion **R** is known as the resource invariant: it must be true at the beginning of **T** and at the end of every atomic region within **T**. Conversely, **R** may be assumed true at the end of the whole block **T** , and at the beginning of every atomic region within it. Note that in this diagram the place is expected to store the token between successive executions of the atomic regions.

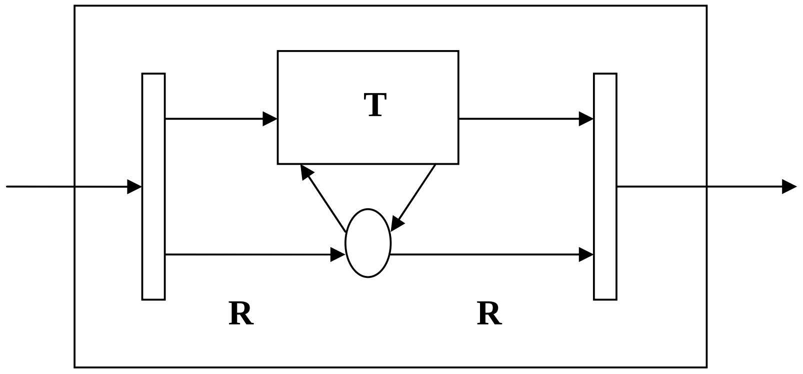


Figure 6.4. Resource Declaration: resource R in T

The explicit statement of a resource invariant permits a very necessary relaxation of the restriction that the assertions used within the threads of **T** may not refer to the values of the variables of the shared resource, for fear that they are subject to concurrent update by concurrent threads. The relaxed restriction states that all of the assertions private to a thread (initial, internal or final) may mention the values of the shared resource, but only in a way that is tolerant of interference. This means that the local assertions of each thread must also be an invariant of every atomic region that may be invoked by the other threads.

A direct application of this proof rule would require proof of each thread to know all the internal assertions in every other thread -- a serious violation of the principal of locality of proof. A stronger but more modular condition is that each thread must prove locally that all its assertions are invariant of any section of code **X** that leaves **R** invariant. The details of the formalisation will not be elaborated here.

The definition of concurrent composition given in Figure 6.2 applies to fragments of ordinary program with a single entry and a single exit. Our final task is to apply the same idea to compensable transactions, with many more entries and exits. The basic definition of concurrency of transactions introduces a transition to fan out each entry of the concurrent block to the two (or more) like-named entries of the components; and similarly, it introduces a transition to fan in the like-named exits of the components to relevant exit of the whole composition (Figure 6.5). This means that the compensations of concurrent transactions will also be executed concurrently in the same way as their forward actions.

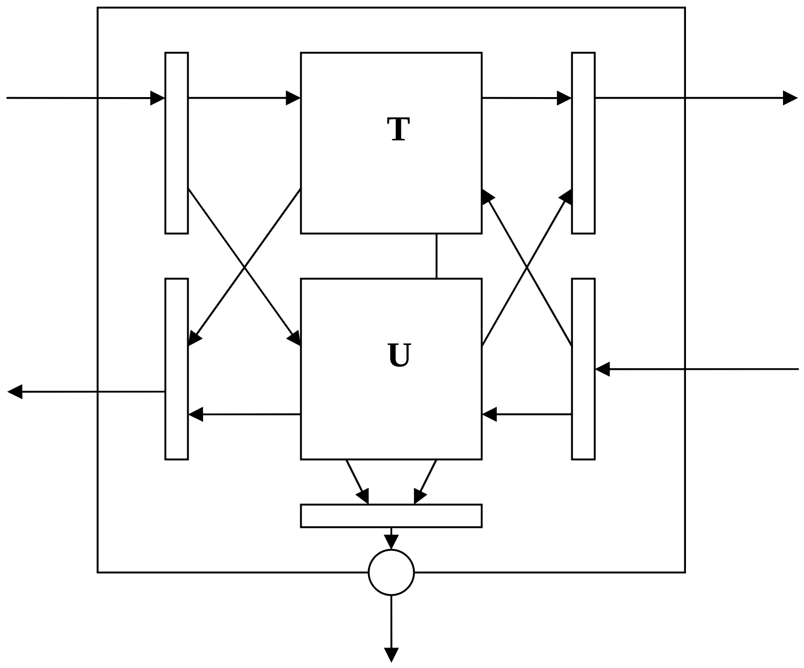


Figure 6.5. [T || U] as a transaction

This scheme works well, provided that both components agree on which exit to activate on each occasion -- either they both finish, or they both fail, or they both throw. The availability of a shared resource enables them to negotiate an agreement as required. However, if they fail to do so, the result is deadlock, and no further action is possible. It may be a good idea to complicate the definition of concurrency of transactions to avoid this unfortunate possibility automatically, by doing something sensible in each case. Four additional transitions are needed to implement the necessary logic.

(1) if one component **T** finishes and **U** fails, these two exits are fanned in by a transition, whose exit leads to the *failback* of the successful **T .**
(2) Similarly, if **U** finishes and **T** fails, the *failback* of **U** is invoked.
(3) In the case that **T** performs a throw but **U** does not, the whole construction must throw. This involves connecting **U**'s *finish* and *fail* exits via a place to a transition that joins it with the *throw* exit of **T .**
(4) A similar treatment deals with the case that **U** performs a throw.

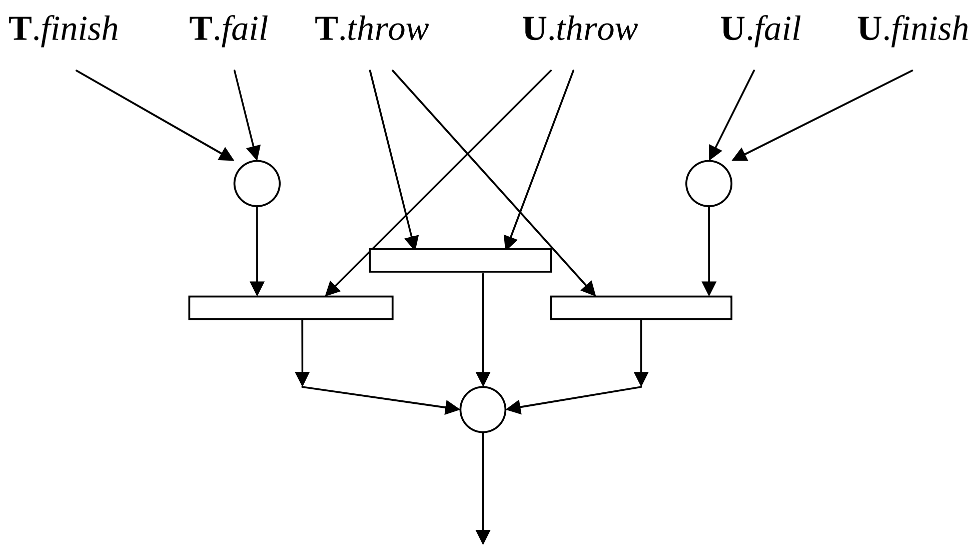


Figure 6.6. Network for *throw*.

Figure 6.6 shows the network controlling activation of the throw exit of [**T** || **U** **].** It is easy to calculate that the correctness condition of the network is

$(\mathbf{T}.\mathit{finish} \;\lor\; \mathbf{T}.\mathit{fail}\,) \land \mathbf{T}.\mathit{throw} \;\lor\; \mathbf{T}.\mathit{throw} \land \mathbf{U}.\mathit{throw} \;\lor$
$\mathbf{U}.\mathit{throw} \land (\mathbf{U}.\mathit{fail} \;\lor\; \mathbf{U}.\mathit{finish})$

7. Conclusion.

This paper gives a simple account using Petri nets of long-running transactions with compensations. The account is also quite formal, in the sense that the nets for any transaction composed solely by the principles described can actually be drawn, programmed and executed by computer. The assertions on the arrows give guidance on how to design correctness into a system of transactions from the earliest stage. The correctness principle for places and transitions serves as an axiomatic semantics, and shows how to prove the correctness of a complete flowchart in a modular way, by proving the correctness of each component and each connecting arrow separately. Thus we have given a unified treatment of both an operational and an axiomatic semantics for compensable, composable and nestable transactions. Simple cases of concurrency can also be treated, but more work, both theoretical and experimental, is needed to deal with more general cases.

The more surprising ideas in this article are (1) use of the precondition of a transaction as the criterion of adequacy of an approximation to the initial state that the compensation should reach (there are many more complicated ways of doing this); and (2) the suggestion of separation logic as an appropriate language for annotating the transitions of a concurrent Petri net.

The deficiencies of this article are numerous and obvious. There are no transition rules, no deductive systems, no algebraic axioms, no denotational semantic functions, no proofs and no references. There is far more work still to be done by anyone sufficiently interested in the subject.

Acknowledgements

The ideas and presentation of this paper have been greatly improved by the helpful comments of:

Michael Butler, Ernie Cohen, Tim Harris, Niels Lohman, Jay Misra, Eliot Moss, Matthew Parkinson, Simon Peyton Jones, Viktor Vafeiadis.