Dependency-based Criteria for Testing Web Services Transactional Workflows

Rubén Casado, Javier Tuya
Department of Computing
University of Oviedo
Asturias, Spain
rcasado@lsi.uniovi.es, tuya@uniovi.es

Claude Godart
LORIA laboratory
University of Lorraine
Nancy, France
claude.godart@loria.fr

***Abstract*— Transactions are a key issue to develop reliable web service based applications. The advanced models used to manage this kind of transactions rely on the dependencies between the involved activities (subtransactions). Dependencies are constraints on the processing produced by the concurrent execution of interdependent subtransactions. Existing works use formal approaches to verify the consistency and correctness of dependencies in web service transactions, but there are no works about testing their implementation. This paper identifies and defines a set of possible dependencies using logical expressions. These expressions define the preconditions to be fulfilled for executing the subtransactions primitive tasks (begin, commit and abort). By using those conditions, we propose a family of test criteria based on control-flow for checking the dependencies between subtransactions. The test criteria provide guidance for test case generation in order to specifically test the implementation of web service subtransactions dependencies.**

Web service testing; transactions; dependencies;

I. INTRODUCTION

Transaction management is a key technology to build efficient and reliable distributed applications. A transaction is defined as a set of operations of an application such that all the operations achieve a mutually agreed outcome. The conventional way for achieving such outcome is the enforcement of the Atomicity, Consistency, Isolation and Durability (ACID) properties which set forward four goals that every transaction management system must ensure. In Web Services (WS) environment the management of transactions is complex as it involves heterogeneous and autonomous services which are loosely coupled, can have long duration and are distributed across the Internet. This scenario forbids the use of locks on resources, and hence makes roll-back activities unsuitable. Various Advanced Transaction Models (ATM) [1] have been proposed for WS. These models mainly relax the strict atomicity and isolation policy of ACID and use a compensation-based policy to achieve an agreed outcome. Each subtransaction has associated a compensatory action that undoes, from a semantic point of view, the action committed by the subtransaction.

A WS transaction comprises a group of a smaller and partially independent subtransactions executed by different WS. To coordinate the execution of the various subtransactions, a set of relationships called *subtransaction dependencies* are specified among them. Dependencies are constraints on the processing produced by the concurrent execution of interdependent subtransactions. Subtransaction dependencies represent a key component in ensuring the flexibility required to support exceptions, alternatives, compensations and so son, which all are the basis of the ATM.

Existing works [2, 3] have addressed the verification of the dependencies model in WS transactional compositions. In these works, the authors propose a formal approach to verify the consistency and correctness between the activities. However, this does not assure that the implementation satisfies the property since there is no formal link between the design model and their implementation. Thus, we cannot ensure that the software fulfills those constraints since the implementation phase may include faults.

Testing is the process of exercising software to determine whether it satisfies specified requirements. Despite some works have been recently published about testing WS transactions [4], there are no approaches focusing on the dependencies [5]. This work proposes a method for defining and testing subtransactions dependencies in WS transactions. Contributions of this paper are twofold. Firstly we identify and define a set of possible dependencies using logical expressions. A set of conditions for beginning, completing and aborting (called subtransactions primitive tasks) are derived from the logical expressions. Secondly we propose a family of test criteria, based on control-flow, for checking the dependencies between subtransactions. The test criteria provide guidance for test case generation in order to specifically test the implementation of web service subtransactions dependencies.

The rest of the paper is organized as follows. Section II defines the different dependencies that can be found in a WS transaction. Our approach formally defines, for each subtransaction, three set of conditions (*BeginCond*, *CommitCond*, and *AbortCond*) using logical expressions. Section III presents a family of dependency-based test criteria by using the conditions derived from the dependencies. Those criteria (partially inspired on control-flow testing criteria [6-8]) are based on two concepts: which primitive tasks are the tests focused on and how the conditions are exercised. In order to show the use of our approach, an example is presented in Section IV. Finally, conclusions and future work are presented in Section V.

II. WS TRANSACTION DEPENDENCIES MODEL

WS is a technology for automating Internet-based interactions. Enterprises are able to outsource their internal business processes as services and make them accessible via the web. Then they can dynamically combine individual services to provide new value-added process. A web service transaction (*wT*) is a conglomeration of existing WS working in tandem to offer an agreed combined outcome. The business process modeled as a *wT* is composed by a set of activities (subtransactions) and a set of relationships (dependencies) between such activities. Each activity (e.g. to book a flight) is executed by an individual web service. The dependencies specify how services are coupled and how the behavior of certain services influences the behavior of other services. So we define a web service transaction as $wT =< S, D >$ where $S = \{s_1, \ldots, s_n\}$ is a set of subtransactions and $D = \{d(s_a, s_b)_1, \ldots, d(s_w, s_z)_m\}$ is a set of dependencies between the subtransactions.

Any subtransaction s_i has a set of primitive tasks that we assume are executed as atomic actions:

- $B(s_i)$: The subtransaction s_i begins executing.
- $C(s_i)$: The subtransaction s_i successfully commits.
- $A(s_i)$: The subtransaction s_i aborts.

An abortion may occur due to either a fault during the execution or an explicit cancellation. When a subtransaction aborts, its compensatory action will be executed if it exists. In our model, a compensatory action is defined as another subtransaction part of the same *wT*. The original subtransaction and their compensatory action are, therefore, related by concrete dependencies as is shown later.

A. Dependencies

Each dependency $d(s_x, s_y)$ defines a relationship between two subtransactions s_x and s_y. The formal definition of the possible dependencies is presented below. The dependencies are divided in three groups (necessary, sufficient, and composite) according to their constraints:

Necessary conditions dependencies: In order to be able to execute any primitive task P, a subtransaction s_y may require the execution of other primitive task Q of a subtransaction s_x. So s_y cannot execute P until s_x has executed Q. Formally, $P(s_y) \Rightarrow P(s_x) < Q(s_y)$. These dependencies are labeled as $abc - on - xyz$ (abbreviated as *ax*) where $abc, xyz \in \{begin, commit, abort\}$. Due to there are three different primitive task and all combinations are possible, nine dependencies are defined as is shown in Table I. For example *begin-on-begin* dependency, $bb(s_x, s_y)$, specifies that the beginning of s_x is a necessary condition to enable the beginning of s_y.

Sufficient conditions dependencies. The execution of any primitive task P of a subtransaction s_x may force the execution of another primitive task Q of a subtransaction s_y. So if s_x executes P, then s_y also executes Q. Formally, P $(s_x) \Rightarrow Q(s_y)$. These dependencies are labeled as $force\ abc - on - xyz$ (abbreviated as *fax*). The nine possible dependencies of this kind are presented in Table II. For example *force begin-on-abort* dependency, $fba(s_x, s_y)$, defines that if s_x abort then s_y has to begin.

Composite dependencies. This group is composed by the dependencies where more than one relationship are taken in account. They are shown in Table III.

TABLE I. NECESSARY CONDITIONS DEPENDENCIES

	Begin	Commit	Abort
Begin	$bb(s_x, s_y)$	$bc(s_x, s_y)$	$ba(s_x, s_y)$
Commit	$cb(s_x, s_y)$	$cc(s_x, s_y)$	$ca(s_x, s_y)$
Abort	$ab(s_x, s_y)$	$ac(s_x, s_y)$	$aa(s_x, s_y)$

TABLE II. SUFFICENT CONDITIONS DEPENDENCIES

	Begin	Commit	Abort
Begin	$fbb(s_x, s_y)$	$fbc(s_x, s_y)$	$fba(s_x, s_y)$
Commit	$fcb(s_x, s_y)$	$fcc(s_x, s_y)$	$fca(s_x, s_y)$
Abort	$fab(s_x, s_y)$	$fac(s_x, s_y)$	$faa(s_x, s_y)$

TABLE III. COMPOSITE DEPENDENCIES

Name	Description	Definition	Example
Weak commit dependency, $wc(s_x, s_y)$	If both s_x and s_y commit, then the commitment of s_x precedes the commitment of s_y.	$C(s_x) \Rightarrow \{C(s_y) \Rightarrow [C(s_x) < C(s_y)]\}$	*If a paper is accepted in a conference then it was sent before the deadline*
Weak abort dependency, $wa(s_x, s_y)$	If s_x aborts and s_y has not been committed, then s_y aborts	$A(s_x) \Rightarrow \{\neg[C(s_y) < A(s_x)] \Rightarrow A(s_y)\}$	*If the user cancels the information request process, the query is not sent to the database*
Termination dependency, $t(s_x, s_y)$	S_y cannot commit or abort until s_x either commits or aborts	$C(s_y) \vee A(s_y) \Rightarrow C(s_x) \vee A(s_x)$	*The final outcome of a process cannot be sent until other process has finished*
Exclusion dependency, $e(s_x, s_y)$	Only one of both s_x and s_y can commit	$[C(s_x) \Rightarrow A(s_y)] \wedge [C(s_y) \Rightarrow A(s_x)]$	*When two hotel providers have been queried, only one can confirm the reservation*
Strong exclusion dependency, $se(s_x, s_y)$	One of both s_x and s_y must commit	$[A(s_x) \Rightarrow C(s_y)] \wedge [A(s_y) \Rightarrow C(s_x)]$	*If there are two possible means of transport, one of them has to be booked for finishing the travel reservation*

B. *Modeling wT using dependencies*

Using the above dependencies we can define aspects related to the management of the transactional process. A *compensatory action* associated to a subtransaction is defined as two dependencies fca and ba. A s_x *replaceable* by s_y can be defined as a dependency $e(s_x, s_y)$, $se(s_x, s_y)$ or a combination of both, depending of the specific context.

Control flow patterns [9], such as *AND-join*, *AND-split*, *OR-join*, *XOR-split, parallel-overlapping, parallel-including* and so on, can be modeled with these dependencies.

AND-join pattern defines that a group of subtransactions have to execute a primitive task before another(s) subtransaction(s) can execute a primitive task. Since it defines necessary conditions to execute a primitive task related to the execution of others subtransactions' primitive task, it is modeled as a set of *necessary conditions dependencies*. For example $bc(s_x, s_z)$ and $bc(s_y, s_z)$ define a *AND-join* pattern between s_x, s_y, s_z where the commitment of s_x, s_y is needed to begin s_z.

OR-join pattern defines a relationship between a group of subtransactions, say s_x, s_y, and another one, say s_z. The execution of the primitive task of any subtransaction s_x, s_y is a sufficient condition to execute the primitive task of s_z. So this pattern is modeled as two *sufficient conditions dependencies* $fbc(s_x, s_z)$ and $fbc(s_y, s_z)$

AND-split pattern defines that once a subtransaction has executed a primitive task, another(s) subtransaction(s) can execute a primitive task. A common use is the serial execution, defined as $bc(s_x, s_y)$, where the subtransaction s_y has to wait until s_x has committed before it can begin.

XOR-split pattern defines a relationship between a group of subtransactions, say s_x, s_y, and another one, say s_z. This relationship specifies that one and only one subtransaction must commit in order to enable s_z to begin. According to the definition, *XOR-split* pattern is defined by a *composite dependency* $e(s_x, s_y)$ and two necessary conditions dependencies $fbc(s_x, s_z)$ and $fbc(s_y, s_z)$.

Two different subtransactions, say s_x, s_y, follow the *parallel overlapping* pattern if and only if the begin of s_x precedes the begin of s_y, the begin of s_y precedes the commitment of s_x, and the commitment of s_x precedes the commitment of s_y. This pattern is defined as three dependencies (s_x, s_y), $cb(s_y, s_x)$ and $cc(s_x, s_y)$. In a similar way, they follow the *parallel including* pattern if and only if the begin of s_x precedes the begin of s_y but the commitment of s_y precedes the commitment of s_x. This pattern is defined as two dependencies $bb(s_x, s_y)$ and $cc(s_y, s_x)$.

C. *From a business process to primitive tasks relationships.*

A business process can be modeled in terms of primitive tasks relationships. Let assume as example the WS transaction depicted in Figure 1.

The initial step is to define the subtransactions involved in the process. According to the figure, we partially define the process as $wT = \{S, D\}$, $S = \{s_0, s_1, s_2, s_3, s_4\}$.

The next step is to identify the control flow patterns (e.g. *AND-split*) and the transaction management aspects (e.g. *replaceable* subtransactions). The example shows a workflow where s_0 is the first subtransaction to be executed. When s_0 has committed, s_1 and s_2 can begin (*AND-split*). Both s_1 and s_2 are required to commit before s_3 can begin (*AND-join*). If s_1 is aborted after it had committed, it is necessary to execute s_4 to undone its action (*compensatory action*, denoted by the broken line). Those relationships are modeled using the dependencies as had been shown before. So we define the set of dependencies as $D = \{bc(s_0, s_1), bc(s_0, s_2), bc(s_1, s_3), bc(s_2, s_3), fca(s_1, s_4), ba(s_1, s_4)\}$

Logical conditions are specified tailoring the dependencies. They define a logical expression that fire a primitive task once is evaluated as true. In other words, they specify a precondition to be enforced before the subtransaction can execute the task. $BeginCond(s_i)$ defines the logical expression, derived from s_i´s dependencies, that controls the subtransaction s_i beginning. It is structured as $BeginCond(s_i) = (N_1 \wedge ... \wedge N_i) \vee (S_1 \vee ... \vee S_j)$, where *N* is a necessary condition and *S* a sufficient condition. In a similar way we can define $CommitCond(s_i)$ and $AbortCond(s_i)$. In this way, the last step in the business process modeling is to define the $BeginCond$, $CommitCond$ and $AbortCond$ expressions for all the subtransactions. To define those expressions is necessary to check all the dependencies where the primitive task is involved. If the dependency defines a necessary condition, it will be added to the left part of the expression (N_{i+1}, linked by $\wedge$). If it is a sufficient condition, it will be added to the right part of the expression (S_{j+1}, linked by $\vee$). The logical expressions for the example are presented in Table IV. The symbol * means that there are no conditions, in other words, the logical expression is always true.

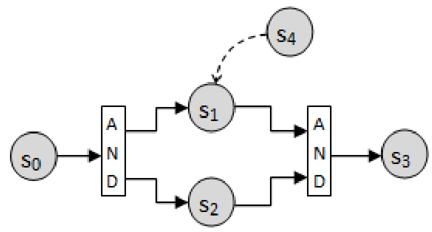

Figure 1. WS transaction example

TABLE IV. BOOLEAN EXPRESSIONS IN THE EXAMPLE

	$\mathbf{BeginCond(s_i)}$	$\mathbf{CommitCond(s_i)}$	$\mathbf{AbortCond(s_i)}$
s_0	*	*	*
s_1	$C(s_0)$	*	*
s_2	$C(s_0)$*	*	*
s_3	$C(s_1) \wedge C(s_2)$	*	*
s_4	$A(s_1)$	$A(s_1)$	*

III. DEPENDENCY-BASED TESTING

The main goal of this work is to define test criteria for testing the dependencies. We base our approach on the subtransactions primitive tasks relationships. A test criterion is defined as a set of rules that impose test requirements and must be fulfilled by the test cases. A coverage criterion provides guidance for tests definition making this process more efficient and effective. Many test coverage criteria have been proposed such as path coverage, branch coverage, data flow coverage and so on [7]. These criteria are applied over some kind of model of the software under test. For example path coverage can be used on a graph that represents the states and transitions of a software component. We define test criteria to be applied on the dependencies model explained in Section II.

We propose a set of criteria based on two primitive set of criteria: *task-based* and *conditions-based*. *Task-based* refers to the primitive task(s) that are checked in the subtransactions. *Conditions-based* refers to the criteria used to check the conditions that compose the logical expressions $BeginCond$, $CommitCond$ and $AbortCond$. Finally, these two primitive criteria are combined to define a family of test criteria.

A. Task-based criteria.

They are regarding the subtransactions primitive tasks to be exercised. Three criteria are defined:

All-begin criterion *(ABC)*: All the subtransactions must begin at least once.

All-commit criterion *(ACC)*: All the subtransactions must commit at least once.

All-commit-abort criterion *(ACAC)*: All the subtransactions must commit and abort at least once.

ACC subsumes *ABC* since any subtransaction needs to begin before committing. Obviously *ACAC* includes *ACC* and, therefore, also include *ABC*. A more exhaustive criterion requires more primitive tasks to be executed and therefore, a higher effort testing process.

Let define a test suite as $T = \{tc_1, \ldots, tc_n\}$, where each tc_i is a test case that describes which primitive tasks have to be executed (and which not) in an execution of a web transaction $wT = \{S, D\}$ .We can formally the previous criteria as follow:

T satisfies the *all-begin criterion* for wT if $\forall s_i \in S$, $\exists tc_j \in T /\ BeginCond(s_i) = true$.

T satisfies the *all-commit criterion* for wT if $\forall s_i \in S$, $\exists tc_j \in T /\ CommitCond(s_i) = true$.

T satisfies the *all-commit-abort criterion* for wT if $\forall s_i \in S$, $\exists tc_j \in T /\ CommitCond(s_i) = true \wedge \exists tc_k \in T /\ AbortCond(s_i) = true$.

B. Conditions-based criteria.

They are used to check the conditions that compose the logical expressions $BeginCond$, $CommitCond$ and $AbortCond$:

Decision criterion *(DC):* Every logical expression has taken true and false outcome at least once.

Decision/Condition criterion *(DCC):* Every logical expression has taken true and false outcome and all conditions in each logical expression have taken true and false outcome at least once.

Modified condition/decision coverage *(MCDC)* [8]: Every logical expression has taken true and false outcome at least once, all conditions in each logical expression have taken true and false outcome at least once, and each condition has been shown to independently affect the logical expression´s outcome (both true and false).

DCC subsumes *DC* and *MCDC* subsumes both *DC* and *DCC*. In the same way as *task-based* criteria, a deeper criterion requires a higher testing effort.

These criteria are formally defined as follow. Let define a transaction $wT = \{S, D\}$, a test suite $T = \{tc_1, \ldots, tc_n\}$ and a logical expression $E \in$ {*BeginCond, CommitCond, AbortCond*}.

T satisfies *DC* for *wT* if $\forall s_i \in S$, $\exists tc_j \in T /\ E(s_i) = true \wedge \exists tc_k \in T /\ E(s_i) = false$.

T satisfies the *DCC* for *wT* if $\forall s_i \in S$, $(\exists tc_j \in T /\ E(s_i) = true \wedge \exists tc_k \in T /\ E(s_i) = false) \wedge (\forall cond \in E(s_i), \exists tc_l \in T /\ cond = true \wedge \exists tc_l \in T /\ cond = false)$

T satisfies the *MCDC* for *wT* if $\forall s_i \in S$, $(\exists tc_j \in T /\ E(s_i) = true \wedge \exists tc_k \in T /\ E(s_i) = false) \wedge (\forall cond \in E(s_i), \exists tc_o \in T /\ E(s_i) = true \Rightarrow (\neg cond \Rightarrow E(s_i) = false) \wedge \exists tc_p \in T /\ E(s_i) = false \Rightarrow (\neg cond \Rightarrow E(s_i) = true)$

C. Dependency-based criteria.

Combining both primitive criteria, we define a family of criteria for testing dependencies in web services transactions. For each *task-based* criteria any *conditions-based* criteria can be applied. So we define nine criteria labeled as *T-C* where *T* is a *task-based* criterion and *C* is a *condition-based* criterion. *T* defines what primitive task will be exercised and, therefore, what logical expressions will be used. *C* defines what criterion will be used to exercise the conditions in such logical expressions. The proposed criteria are *ABC-DC*, *ABC-DCC*, *ABC-MCDC*, *ACC-DC*, *ACC-DCC*, *ACC-MCDC*, *ACAC-DC*, *ACAC-DCC*, *ACAC-MCDC*.

For example, in the *ACC-DCC* criterion, *ACC* requires all the subtransactions to commit, so the logical expressions to be used are $CommitCond(s_i)$. *DCC* requires all the conditions in each logical expression to take true and false outcome at least once. So *ACC-DCC* criterion is defined as follow:

ACC-DCC: All the subtransactions must commit at least in one test case, all subtransaction must not commit at least in other another test case and all conditions in the committing logical expression have taken true and false outcome at least in one test case. Formally, let $wT = \{S, D\}$, and $T = \{tc_1, \dots, tc_n\}$, $\forall s_i \in S$, $(\exists\ tc_j \in T / CommitCond(s_i) = true \wedge \exists\ tc_k \in T / CommitCond(s_i) = false) \wedge (\forall\ cond \in CommitCond(s_i),\ \exists\ tc_l \in T /\ cond = true \wedge \exists\ tc_l \in T /\ cond = false)$

In the same way as is shown for *ACC-DCC*, the rest of dependency-based criteria can be defined.

IV. Example

In order to show the complementarity of our approach with existing verification-based techniques, we will use the example presented in [3]. In that work, the authors presented a method to ensure the correctness of WS compositions. Here, we use the test criteria to check those identified requirements in the design phase regarding the implementation.

The example is an application dedicated to the online purchase of personal computer (OCP). This application is carried out by a composite service as illustrated in Figure 2. We assume the process design has been correctly verified so our goal is to find faults in the implementation. Services involved in this application are: the Customer Requirements Specification (CRS) service used to receive the customer order and to review the customer requirements, the Order Items (OI) service used to order the computer components if the online store does not have all of it, the Payment by Credit Card (PCC) service used to guarantee the payment by credit card, the Computer Assembly (CA) service used to ensure the computer assembly once the payment is done and the required components are available, and the Deliver Computer (DC) service used to deliver the computer to the customer (provided either by Fedex (DF) or TNT (DT)).

The whole purchase process is identified as a WS transaction. As is identified in [3], several dependencies are necessary between the subtransactions. Some dependencies are directly defined by the flow patterns (e.g. AND-split pattern). On the other hand, some dependencies are required due to the relationship between subtransactions. If OI service is does not complete, the payment service PCC has to be compensated. In the same way, OI is compensated by cOI since if PCC fails, the order must be undone. Also there is a dependency between the delivery services since only one and only one must commit. The WS transaction is modeled as is shown in Section II.B. The logical expressions derived from the dependencies in the OCP example is shown in Table V.

$wT_{OCP} = \{S_{OCP}, D_{OCP}\}$

$S_{OCP} = \{CRS, OI, cOI, PCC, cPCC, CA, DF, DT\}$

$D\_OCP = \{bc(CRS, OI),\ bc(CRS, PCC),\ bc(OI, CA),\ bc(PCC, CA),\ bc(CA, DF),\ bc(CA, DT),\ fca(OI, cPCC),\ bc(PCC, cPCC),\ fca(PCC, cOI),\ bc(OI, cOI),\ e(DF, DT),\ fe(DF, DT)\}$

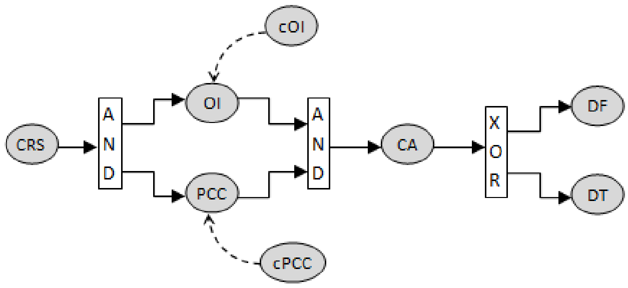

Figure 2. OCP application

TABLE V. Logical Expressions in OCP Application

	$BeginCond(s_i)$	$CommitCond(s_i)$	$AbortCond(s_i)$
CRS	*	*	*
OI	$C(CRS)$	*	*
cOI	$A(PCC) \wedge C(OI)$	$A(PCC)$	*
PCC	$C(CRS)$	*	*
cPCC	$A(OI) \wedge C(PCC)$	$A(OI)$	*
CA	$C(OI) \wedge C(PCC)$	*	*
DF	$C(CA)$	*	$C(DT)$
DT	$C(CA)$	*	$C(DF)$

A. Use of test criteria

Since there is an infinite number of possible test cases, it is necessary to define a subset of all possible tests. A test criterion will provide guidance for test cases generation. A test case is a specific way of executing the application in order to cover one or more requirements defined by the test criterion. To our field, such requirements are the value of the conditions that compose the logical expressions. So a test case describes which primitive tasks have to be executed (and which not) in an execution of a web transaction.

Once the dependency-based criterion is chosen, the next step is to systematically apply it over the model. Let assume we want to apply *ABC-MCDC* for OCP application. The task-based (*ABC*) criterion specifies that all subtransactions have to begin at least in one test case and not to begin in at least another different test case, so the *BeginCond* expressions will be used. Since the condition-based criterion is *MCDC*, every condition of each *BeginCond* expression has to take a true outcome in at least one test case and a false outcome in at least another different test case and, in both case, the value has been shown to affect the final expression´s outcome. For example the *BeginCond* for CA subtransaction is $BeginCond(CA) = C(OI) \wedge C(PCC)$, as is shown in Table V. *MCDC* criterion applied over $BeginCond(CA)$ require one test case where the expression takes the false outcome due to $C(PCC)$ is false. $C(PCC)$ may be false because it has not begun. In order to make true *C(OI)*, it requires CRS subtransaction to commit. So the conditions are defined (*T*=true, *F*=false) as *B(CRS)=T,*

C(CRS)=T, B(OI)=T, C(OI)=T, B(PCC)=T. It defines a situation where CRS receives and successfully reviews the customer requirements and then contacts with OI and PCC. While the OI service achieves correctly its goal (begin and commit the subtransaction), the PCC service does not execute its subtransaction. In this way, according to the defined dependencies, CA service must not begin and thus, the rest of process is not executed. The rest of test case according to the criteria can be defined in the same way.

The application of the proposed test criteria allows deriving *positive* and *negative* test cases.

A *positive test case* exercises the application in a right way, in other words, according to the specification. For example the test scenario TC1 identified in Figure 3 achieved using *ABC-DC* criterion. Dash means that it does not matter what is the value. The test scenario defines the following execution: The Customer Requirements Service (CRS) receives y reviews successfully the customer order. The Order Items service (OI) has successfully ordered the required items and the payment has been successfully done using the Payment service (PCC). These two actions have been begun in parallel. Later, the computer is successfully assembled. Finally the two delivery services are notified to check their availability to be used. This test case could detect failures of extra dependency implementation; for example, if OI waits to order the items until PCC has charged the payment, the whole process will take longer time keeping the resources busy and maybe rejecting new orders where they are actually free.

A *negative test case* exercises the application in a wrong way. It means that the execution tries to break the specification. This kind of test case can detect fault of dependencies implementation omission. For example the test scenario TC2 identified in Figure 3, achieved using the *ABC-DC* criterion too. This test case tries to order and to charge without reviewing the customer requirements. If the scenario can be executed, a failure will be detected: the constraints of successfully committing of CRS before OI and PCC can begin are not implemented. So a purchase of incompatible items for a personal computer can be allowed.

V. Conclusions and Future Work

Transactions are key issue to ensure consistency in WS compositions. Since the ACID properties became unsuitable in a loosely coupled world of services, new models have been proposed to deal with the problem of achieving an agreed outcome without locking the resources.

TC1	*Begin*	*Commit*	*Abort*
CRS	T	T	F
OI	T	T	F
cOI	F	F	F
PCC	T	T	F
cPCC	F	F	F
CA	T	T	F
DF	T	–	–
DT	T	–	–

TC2	*Begin*	*Commit*	*Abort*
CRS	T	F	F
OI	T	T	F
cOI	F	F	F
PCC	T	T	F
cPCC	–	–	–
CA	–	–	–
DF	–	–	–
DT	–	–	–

Figure 3. Test design

These advanced models discompose the transaction in smaller independent subtransactions and rely on strict dependencies between them.

The literature presents many works about dependencies verification at design phase and this paper complements such works addressing the verification of the implementation with regard to the specification. In this paper we have presented a set of test criteria to guide the test case generation. The criteria are based in the logical conditions defined by the dependencies that manage the execution of the subtransactions primitive tasks. Our work is focused on failure detection of the dependency requirements after the implementation phase. So this work is a complementary approach to the formal verification-based approach proposed in [3]. Whereas the formal verification checks if the specification is wrong, our approach allows detecting if the implementation does not match the specification.

Although the proposed criteria allow deriving test cases from a specification, more research is needed to improve the method. A deeper analysis will contribute to identify relationships between the test effort of each criteria and its effectiveness. Also it is necessary to determine the adequacy of each criterion according to the different possible dependencies.

Acknowledgment

This work was partially funded by the Department of Science and Technology (Spain) and ERDF funds under the National Program for Research, Development and Innovation, project Test4DBS (TIN2010-20057-C03-01), and the grant BES-2008-004355.

References

[1] A. K. Elmagarmid, Database transaction models for advanced applications, Morgan Kaufmann Publishers, 1992.

[2] W. Gaaloul, M. Rouached, C. Godart, M. Hauswirth, “Verifying composite service transactional behavior using event calculus,” in Proceedings of the OTM Confederated Int. Conf. on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I, Vilamoura, Portugal, 2007, pp. 353-370.

[3] S. Bhiri, O. Perrin, and C. Godart, “Ensuring required failure atomicity of composite Web services,” in Proceedings of the 14th international conference on WWW, Chiba, Japan, 2005, pp. 138-147.

[4] R. Casado, J. Tuya, and M. Younas, “Testing Long-Lived Web Services Transactions Using a Risk-Based Approach,” in Proceedings of the 10th International Conference on Quality Software, 2010, pp. 337-340.

[5] G. Canfora and M. Penta, "Service-Oriented Architectures Testing: A Survey," in Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures, ed: Springer-Verlag, 2009, pp. 78-105.

[6] G. J. Myers, The art of software testing, New York :: Wiley, 1979.

[7] H. Zhu, P. A. V. Hall,. "Software unit test coverage and adequacy." ACM Comput. Surv. 29(4): 366-427, 1997

[8] RCTA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, RTCA, Washington, USA, 1992.

[9] S. Bhiri, C. Godart, and O. Perrin, “Transactional patterns for reliable web services compositions,” in Proceedings of the 6th international conference on Web engineering, Palo Alto, California, USA, 2006, pp. 137-144.