Querying and splitting techniques for SBA: a model checking based approach[*]

Yahia Chabane[1], François Hantry[2] and Mohand-Saïd Hacid[2]

[1]University Blaise Pascal Clermont-Ferrand 2, LIMOS UMR 6158, France
[2]University Claude Bernard Lyon 1, LIRIS CNRS UMR 5205, France
`yahia.chabane@isima.fr,`
`{francois.hantry, mohand-said.hacid}@liris.cnrs.fr`

Abstract. Current approaches to fragmentation of services are not business-oriented. They are not based on a real temporal query language, and in general, they return execution traces instead of parts of the process. We propose in this work an approach for fragmentation based on model checking and slicing techniques. Fragmentation is based on business rules expressed in LTL. In our work the fragmentation does not consist in splitting a web service composition in a set of fragments. It is defined as the seeking of a single fragment that contributes to the verification of a business rule.

Keywords: process fragmentation, web services, model checking, business processes, temporal logics

1 Introduction

A Service-Based Application (SBA) is made of a number of possibly independent services, available on the network. The services perform the desired functionalities of the architecture. Such services could be provided by third parties, not necessarily by the owner of the service-based application. A service-based application displays a difference with respect to a component-based application: while the owner of the component-based application also owns and controls its components, the owner of a service-based application does not own, in general, the component services, nor it can control their execution [35].

Let us consider an example of a process of online purchase. The system is implemented as a complex composition of web services. The process is composed by a subset of features: payment, bidding, ordering …. We want to secure the payment process without affecting other functionalities. It is interesting to determine the part of the payment that work on the whole process which can be very difficult when the process is complex. This can be seen as a fragmentation process by answering a

[*] This work is carried out in the framework of the FP7 European project COMPAS GAN 215175, http://www.compas-ict.eu/.

question like "what is the part who manage the payment?". Benefits of fragmentation, among others, are analysis and reusing.

The goal of this work is the fragmentation of a business process by using business rules expressed in linear temporal logic. Given a BPEL composition of web services, we want to determine the part of a process that contributes and ensure the verification of the business rules.

A fragment is a subset of activities of a composition of web services. Fragmentation [34] is the act of creating fragments out of one service composition by applying a fragmentation technique. A fragmentation technique is a method to perform fragmentation according to some fragmentation criteria. The fragmentation criteria may be described in natural language, e.g. “the resulting fragments group the activities according to who executes them”. Fragmentation techniques combine the following two steps [34]:

- Fragment Identification identifies which elements belong to which fragment.
- Fragment Severing removes the elements comprised in a fragment from the service composition, possibly substituting them in the service composition with other elements that were not initially included.

Current techniques of fragmentation can be divided into two classes: query languages for fragmentation and fragmentation for migration, optimization, transaction and performance. A Query language for fragmentation groups slicing [37], LTL [13], BPQL [3] and goal oriented fragmentation [31]. One unexplored problem is temporal slicing. Current approaches are not business oriented. They are not based on real temporal query languages. Fragmentation is generally used for migration and performance, the fragment is not determined regarding business rules and in general, they return execution traces instead of part of the process. To address these limitations, we propose an approach to produce fragments using business queries, this query is based on a temporal logic. Our approach is based on model checking. We try to make a mix of slicing and LTL query language.

Our work is an attempt to define fragment model compositions of web services, the two contributions are:

- Proposal of an approach of fragmentation based on business rules.
- Refinement of a fragment to differentiate between two types of statements in a fragment.

The rest of the paper is organized as follows. We will give definitions to introduce some concepts in the next section. We discuss previous work in section 3. Section 4 is the core of our work. We will define a model of fragmentation and distinguish between two constructs of a fragment. In section 5, we will summarize our approach and present future work.

2 Preliminaries

2.1 Temporal logics

Temporal logics [13] were originally developed to study patterns of truth that depend on the evolution of the world. It provides a formal system for qualitative description and reasoning about how the truth values change over time.

The temporal logic is well suited to describe programs that don't have a final statement as operating systems, which cannot be described using the classical logic. It is used in virtually all aspects of design of concurrent programs.

The basic temporal operators are: *Fp* (sometime *p*) *Gp* (always *p*), *Xp* (next time *p*) and *pUq* (*p* until *q*). Several types of temporal logics have been proposed, the most used ones are the linear temporal logic [13] and branching temporal logic [13].

2.1.1 Linear Temporal Logic (LTL)

In this type of logic, the system is modeled as a sequence of states, the evolution of time is linear, it is considered as discrete. To simplify the clock starts at the initial state which has no predecessor and infinite in the future.

Let *AP* be an underlying set of atomic proposition symbols. We can then formalize the notion of a timeline as a linear time structure $M = (S, x, L)$ [13] where:

- *S*: is a set of states.
- *x (N->S)* : is an infinite sequence of states.
- *L (S->PowerSet(AP))* : is a labelling of each state with the set of atomic propositions in AP true at the state.

This type of temporal logic is mainly used in concurrent programming, the formulas are defined by induction [13]:

- Each atomic proposition *p* is a formula.
- If *p* and *q* are formulae then $p \wedge q$ and $\neg p$ are formulae.
- If *p* and *q* are formulae then $p \cup q$ and X_p are formulae.

Let x^i be the path suffix s_i, s_{i+1}, s_{i+2}… The semantics of LTL can be summarized as:

- $M, x \models p$: in structure *M* formula *p* is true in timeline *x*.
- $x \models p$ if $p \in L(s_0)$, for atomic proposition *p*.
- $x \models p \wedge q$ if $x \models p$ and $x \models q$
- $x \models \neg p$ if it is not the case that $x \models p$
- $x \models p\ U\ q$, read as "*p* until *q*" asserts that *q* does eventually hold and that *p* will hold everywhere prior to *q* : if $\exists j (x^j \models q$ and $\forall k<j\ (x^k \models p))$ $x \models X_p$ if $x^1 \models p$
- X_p, read as "next time *p*" holds now if *p* holds at the next moment.
- F_q, read as "sometimes *q*" or "eventually *q*" and meaning that at some future moment *q* is true : $x \models F_q$ si $\exists j\ (x^j \models q)$
- G_q, read as "always *q*" or "henceforth *q*" and meaning that at all future moments *q* is true : $x \models G_q$ si $\forall j\ (x^j \models q)$.

We say that a formula *p* is satisfiable if there exists a linear time $M = (S, x, L)$ such that: $x \models p$. *M* is a model for *p*.

LTL is used to prove that System (structure *M*) satisfies (is a model) for a period a set of properties (a set of formulae).

2.1.2 Branching temporal logic

The structure of time in this type of logic corresponds to an infinite tree, from a given state, we can have several possible future states. Two temporal operators have been added: A_p (for all future) which means that for all possible paths *p* is true and E_p (For Some future) which means that there is at least one path p which is true.

The temporal structure is formalized as $M = (S, R, L)$ where:

- *S*: is the set of states.
- *R*: is a total binary relation $\subseteq S \times S$.
- *L (S->PowerSet(AP))* : is a labelling which associate with each state s an interpretation *L(s)* of all atomic proposition symbols at state *s*.

M is viewed as a labeled graph (Kripke structure), with *S* as a set of nodes, all arcs *R* and *L* of the labeled nodes.

2.1.3 Formalization of business rules with LTL

A business rule [4] is an assertion that defines an aspect of business. We are interested in this work to a subset of business rules, the rules expressible in LTL [29, 33]. The advantage of using LTL is that much of business rules are based on time which is taken in consideration by the temporal logic. One can for example express business rules of type "*delivery of the product should not exceed two days*" by "$\neg period\_exceed U (product\_delivery \wedge \neg period\_exceed)$".

2.2 Model Checking

The problem of model checking [7] is to check if a temporal structure M defines a model for a temporal formula p. The model checking can create interesting applications for automated verification of concurrent systems.

Model checkers typically have three main components: (1) a specification language, based on propositional TL, (2) a way of encoding a state machine representing the system to be verified, and (3) a verification procedure, that uses an intelligent exhaustive search of the state space to determine if the specification is true or not. If the specification is not satisfied, then most model checkers will produce a counterexample execution trace that shows why the specification does not hold. It is impossible to overestimate the importance of this feature. The counterexamples are invaluable in debugging complex systems [6].

There are three main families of model checking algorithm: Symbolic Model Checking with ordered binary decision diagrams (OBDDs), Partial Order Reduction and Bounded Model Checking with SAT. Details can be found in [6].

Several model checkers has been developed. Examples are: NuSMV which is a symbolic model checker, and SPIN which is a partial order reduction model checker. In this work, we will use the Spin model checker [20] which is more appropriate for concurrent systems [23]. Spin [36] is in the family of Partial Order Reduction model checker.

The specification language underlying SPIN is called PROMELA. The name SPIN was originally chosen as an acronym for Simple PROMELA Interpreter. PROMELA is a specification language for parallel asynchronous systems. It allows describing concurrent systems, especially communication protocols.

2.2.1 Verification of web services

The general principle of the work done in the verification of compositions of Web services is the abstraction of the latter in a formal model, then from such an abstraction to the language of the chosen model checker. The well known model checkers are SPIN and NuSMV.

Among the projects undertaken in the field, we can cite the work of Zhao et al. [42], who proposed a formal model for verification of choreography of web services. They translate choreography to orchestration then to PROMELA for using SPIN model checker. Fisteus et al. [14] propose VERBUS, a framework consisting of three layers: layer process definition, common formal model layer which is based on a transition system, and a layer of verification. The advantage is that the verification is independent of the used model checker and process definition.

In [25] the authors propose a model based on multi-agent systems for verifying temporal and epistemic properties in a composition of web services. They use a special language of description system (ISPL), with a symbolic model checker (MCMAS) dedicated to Multi Agent Systems. They propose an approach to verification of the behavior of a service and knowledge gained during the composition, in contrast to model checkers NuSMV and SPIN which are limited to the verification of temporal modality.

Fu et al. [16] propose a model for the analysis of a BPEL composition of web services. Composition of web services is seen as a pattern of conversations between a set of peers. They define a set of rules to move from BPEL composition to their automata model then from their model to a PROMELA specification. They use SPIN for verification. The authors also introduce the concept of synchronizability as a transformation of an asynchronous communication to a synchronous communication to facilitate verification. They propose a series of conditions that must be met to enable synchronizability. The authors have developed a tool that was used, it allows for the passage of a BPEL composition of web services to PROMELA specification.

2.3 Theorem proving

Theorem provers [15, 21, 26, 27] exists for linear temporal logic. However, since deciding satisfiability is P-space complete, this is in general not tractable. Temporal Theorem prover [11, 15] are less efficient than Model checker. Model checking technics is theoretically also P-space complete, however efficient model checking tools as SPIN or NUSMV have been developed.

Spin is more convenient for concurrent system and is based on automata theory. At a first glance it appears that theorem proving is not convenient for our goal of explanation or fragmentation because of tractability issue. However recently, theorem proving gets new research interest because theorem proving tackles infinite state space system, and is more and more used for producing counterexample [28] and provides a real explanation [17]. Since performance problems still remain in the area of Unbounded Model Checking (to prove property), we let theorem proving for future research. And particularly focus on automata-based model checking techniques.

2.4 Slicing

A program slice consists of the parts of a program that (potentially) affects the values computed at some point of interest [37]. Such a point of interest is referred to as a slicing criterion, and is typically specified by a location in the program in combination with a subset of the program's variables. The task of computing program slices is called program slicing.

A program slice *S* can be defined as a reduced, executable program obtained from a program *P* by removing statements, such that *S* replicates part of the behavior of *P*. Another common definition of a slice is a subset of the statements and control predicates of the program that directly or indirectly affect the values computed at the criterion, but that do not necessarily constitute an executable program. An important distinction is that between a static and a dynamic slice. The former is computed without making assumptions' regarding a program's input, whereas the latter relies on some specific test case [37].

Nanda et al. [30] formally defined the slicing in concurrent systems and propose an algorithm for slicing a concurrent program. They extend the classical models of representation to address new emerging types of dependence in parallelism. The authors propose optimizations to be done to prevent the complexity since it becomes exponential if the number of processes is large.

3 Fragmentation: state of the art and problem

3.1 Definition

A fragment is a subset of the elements of a service composition. Unless specified otherwise, no assumptions are taken on the elements included in a fragment (e.g. on how they relate each other), except there is at least one. Fragmentation is the act of creating a set of fragments from one service composition. Fragmentation techniques are procedure, algorithm or methodology to perform fragmentation according to predefined criteria in order to achieve a certain goal [34].

3.2 Type of fragmentation

Fragmentation techniques greatly differ in which types of process-based service compositions they are applicable to, why they are applied, how they define the fragments, etc. The state of the art lacks consistent terminology and definitions for the properties of the fragments of process-based service compositions and the criteria for classifying the different fragmentation techniques [34].

Mancioppi et al present in [34] criteria for classification techniques of fragmentation process based on web service composition. They are divided hierarchically in main and subcriteria, e.g. the main criterion *What input* is further specialized in *Composition paradigm*, *Modeling language*, *Abstraction level*, *Well-formedness*, and *Self-containment*. The classification is summarized in fig.1.

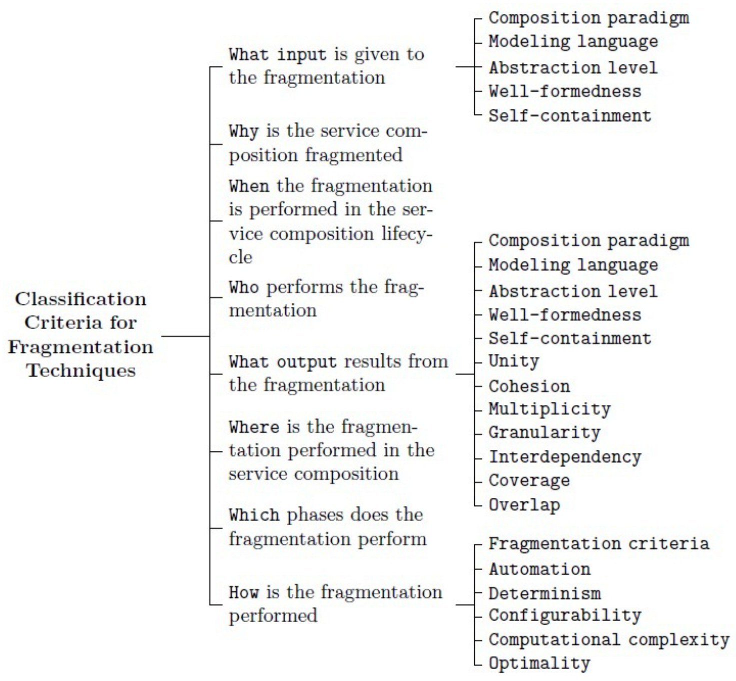

Classification Criteria for Fragmentation Techniques
What input is given to the fragmentation
Why is the service composition fragmented
When the fragmentation is performed in the service composition lifecycle
Who performs the fragmentation
What output results from the fragmentation
Where is the fragmentation performed in the service composition
Which phases does the fragmentation perform
How is the fragmentation performed
Composition paradigm
Modeling language
Abstraction level
Well-formedness
Self-containment
Composition paradigm
Modeling language
Abstraction level
Well-formedness
Self-containment
Unity
Cohesion
Multiplicity
Granularity
Interdependency
Coverage
Overlap
Fragmentation criteria
Automation
Determinism
Configurability
Computational complexity
Optimality

Fig. 1. The classification criteria for fragmentation techniques [34]

Fragmentation techniques are important tools for changing service compositions in response to evolving requirements. However, the lack of a consistent taxonomy for classifying the different fragmentation techniques and the properties of the fragments they produce has hindered their comparison and reuse [34].

3.3 Current techniques

Current techniques of fragmentation can be divided into two broad classes: query languages for fragmenting and fragment for migration, optimization, transaction and performance. Query languages for fragmentation groups slicing, LTL, BPQL and goal oriented fragmentation.

In [34] the authors studied classification techniques of fragmentation They summarized the different approaches in four papers:

Optimal Stratification of Transactions [9]: they treat how to fragment a service composition with transactional properties in order to optimize its costs and non-functional quality aspects. The idea is to divide the service composition into many "connected" global transactions called strata. Strata are fragments of the service composition, each one coordinated by a 2PC protocol. The strata communicate with each other for coordination purposes using persistent message queues.

Towards Identification of Fragmentation Opportunities for Service Workflows with Sharing-Based Independence [22]: they present a fragment identification that can be applied for different purposes such as reuse, optimization of resource utilization and optimization of the non-functional properties of a service orchestration. Sharing-based independence analysis is a general technique that can be applied to both upper and lower layers of software architecture, and consequently, to various parts of the service stack. On the service composition layer, one model different entities used within a workflow as data structures subject to sharing analysis.

Towards Runtime Migration of WS-BPEL Processes [40]: they propose an approach for fragmentation of the process instances that enables the decentralized execution of the process instance by several parties. The decentralized execution of business process instances is a promising approach for enabling flexible reactions to contextual changes at runtime. To do this, the work focuses on the runtime fragmentation of process instances, allowing several (potentially pre-selected) parties to execute a given process instance in a decentralized way. The main goal here is to enable a flexible adaptation of the responsibilities for the execution of the process (in whole or in part) to dynamically changing situations at runtime.

Executing Parallel Tasks in Distributed Mobile Processes [41]: they present an approach that supports the distributed parallel process execution with multiple mobile process participants. In case of a sequential execution of process fragments, the efforts of coordination can often be reduced to a (relatively simple) delegation resp. migration protocol. However, advanced synchronization and coordination mechanisms are required, if parallel process fragments have been distributed to several different parties. If, in addition, shared data objects are used in more than one of these parallel fragments, a separate execution could lead to undesired or wrong results. This contribution considers the concurrent execution of several parallel paths of the process instance by replication of the process description and respective execution of the parallel section of the process by different participants, including synchronization of control flow and data variables.

Among the works done in the area, Khalaf [24] has presented an automatic and operational semantic-preserving decomposition of business processes, in the presence of shared variables, loops, compensation, and fault handling. Their approach has been shown to be interoperable through the use of open standards, as well as transparent. It has also met the goal of not requiring new middleware unless loops and scopes are split, in which case, it requires extensions to existing middleware (i.e. BPEL engine and WS-Coordination framework).

Vanhatalo et al. [38] proposed a technique to focus and speed up control-flow analysis of business process models that is based on decomposition into single-entry-single-exit (SESE) fragments. The SESE decomposition could also be used for other purposes such as browsing and constructing large processes, discovery of reusable subprocesses, code generation, and others. They also proposed a partition of the fragments into various categories, which can be computed fast.

3.4 Problem

The problem of current approaches is that they are not business oriented, they are not based on a real temporal query language. Fragmentation is generally used for migration and performance. The fragment is not determined regarding business rules and in general, the approaches return execution traces instead of part of the process.

4 A business rule base approach to fragmentation

4.1 Preliminaries

Given a web service composition expressed in BPEL, this composition can be an orchestration or choreography, our problem is to break down a web service composition based on a business rule expressed in LTL. That is to say, to find in the service composition a portion that contributes to the verification of the business rule.

The part of the program that helps verify a given property is the cause of non-verification of the negation of the property. The idea is then to prove the non-verification of the negation of a business rule by the generation of a counterexample, then look for the causes of non-verification of the negation of the business rule from this counterexample.

It is easier to prove the non-satisfaction of a property (by the generation of a counterexample), than to prove the satisfaction of a property. Another advantage is the availability of tools for checking an LTL formula. The LTL formulas are well suited to the formalization of some business rules because they treat the concept of time which is the foundation of most business rules.

We define our problem as a problem of causality that was introduced first by Halpern and Pearl [18]. We consider the problem of finding parts that contribute to the verification of a business rule as the one that causes the denial of falsifying business rules.

The formal definition of the concept of causality used in this work is built on the work of Beer et al. [2]. The authors proposed an algorithm to determine from a counterexample such major causes of a malfunction of a system. Halpern et al. [18] have proposed a new definition of causality using structural equations. Eiter et al. [12] investigated the complexity of determining all causes in a binary causal model and Chockler [5] defined the notion of responsibility for verification of a specification. Our work rests on the following two definitions:

Definition of Critical variable [2, 5]: Let *M* be a model, *V* is a finite set of variables, $\vec{u}$ the current context, and η a Boolean formula. Let $(M, \vec{u}) \models \eta$, and *X* a Boolean variable in *M* that has the value *x* in the context $\vec{u}$, and $\overline{x}$ the other possible value (*0* or *1*). We say that $(X = x)$ is critical for η in $(M, \vec{u})$ if $(M, \vec{u}) \models (X \leftarrow \neg x) \neg\eta$. That is, changing the value of *X* to $\neg x$ falsifies η in $(M, \vec{u})$.

Definition of Cause [2, 12, 18]: We say that $X = x$ is a cause of η in $(M, \vec{u})$ if the following conditions hold:

- AC1. $(M, \vec{u}) \models (X = x) \wedge \eta$.
- AC2. There exists a subset $\vec{W}$ of *V* with $X \notin \vec{W}$ and some setting $\vec{w}'$ of the variables in $\vec{W}$ such that setting the variables in $\vec{W}$ to the values $\vec{w}'$ makes $(X = x)$ critical for the satisfaction of η.

In our case, the execution trace (counterexample) can be seen as a binary model. Binary variables of the model are all binary variables of the linear temporal structure representing the trace taken in each state. The concept of time will be taken into consideration. A binary variable *p* to a state s_1 is distinct from *p* to a state s_2. The total number of binary variables in the model will be equal to the multiplication of binary variable in each state by the number of states of the trace.

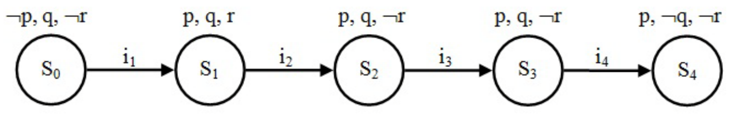


Fig. 2. Example of an execution trace

Let us consider the example of fig.2 showing a linear temporal structure built from an execution trace of a counterexample. Each statement in the trace represents a transition between a state and another. Binary variables on the state of system variables (*price>200* for example) are associated with each state. The structure consists of 5 states and 15 variables. Let a formula $\theta = G(\neg r)$, $\neg r$ is critical for θ at s_1 , because if we reverse *r* in s_1, θ becomes valid, so (r,s_1) is a cause of non verification of θ. Let now the formula $\Omega = G(q \wedge \neg r)$, $\neg r$ in s_1 is not critical for Ω but if we inverse $\neg q$ in s_4 it will become critical, so we can say that *(r, s1)* is a cause for Ω. This illustrates the difference between a critical variable and a cause.

Based on the foregoing definition of cause, the determination of all possible causes of malfunction in a system modeled by a binary model [18] is NP-complete [12], the number of possible cases is exponential. One might also note that in real systems the number of states may be infinite, which complicates the problem.

4.2 Approach

The goal of our work is the fragmentation of the compositions of web services from a business rule expressible in LTL. In other words, given a business rule that can be expressed in LTL, we want to determine the part of a composition of web services that contributes to the verification of the business rule. One major advantage of the fragmentation of the compositions of web services is that it allows a grouping according to the feature on a web service composition. We can also cite other advantages such as simplicity and reuse both in the update and the query.

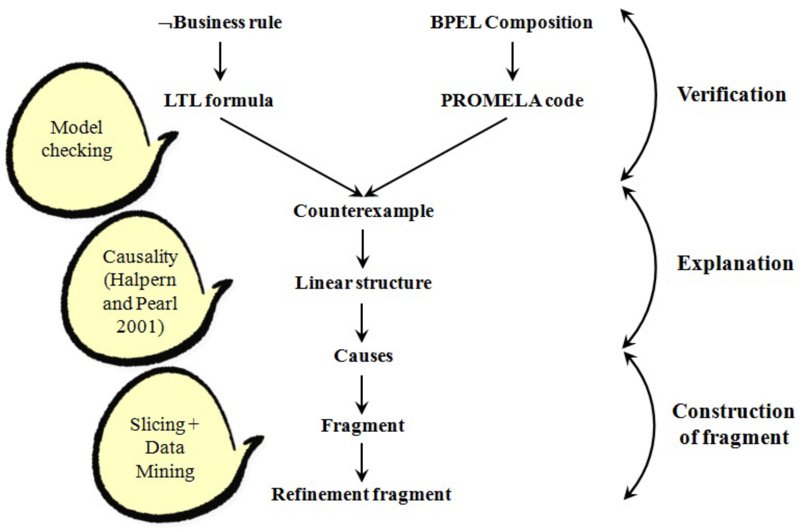


Fig. 3. General principle of fragmentation

The general principle of our approach is illustrated in fig.3, the fragmentation of a web service composition is mainly done through three steps: verification, explanation of the trace and the construction of the fragment. The verification is to show that the negation of the business rule is not checked, this is done by generating a counterexample using the SPIN model checker. The explanation of the trace is to seek the causes of the falsification of the negation of the business rule. The construction of the fragment is to select the statements that contribute to the achievement of the causes identified by the previous step. We will detail each step in the following.

4.3 Verification

The first step of fragmentation is to produce a counterexample to show that the negation of the business rule is not checked. To generate a counterexample we use verification techniques of web services. We use SPIN as a model checker because it is more appropriate to concurrent systems [23]. SPIN takes as input an LTL formula and a model of the system to check expressed in PROMELA.

We consider web service compositions expressed in BPEL, but the entry of SPIN must be specified in PROMELA which is a language for specifying concurrent systems, so we must translate a specification from BPEL to PROMELA in order to use SPIN. We use WSAT (Web Service Analysis Tool) [39] proposed by Fu et al. [16] for the translation from a BPEL to PROMELA.

4.4 Explanation

After generating the counterexample, we must explain and seek the causes of a non verification rule. We explain the causes of the non verification of the negation of the business rule in order to explain the audit of the business rule, that is to say that fragmentation will be on the negation of the formula from which we generated the counterexample. The explanation of the track is the most important step in the process, the quality of fragmentation depends on this step.

In this step, we build a temporal linear structure from a counterexample generated by the model checker. Each statement in the trace represents a transition in the linear structure between a state and another. A same set of binary variables is associated with each state, these variables focus on the state of system variables (for example, *price <200*), a binary variable *p* to a state s_1 is distinct from *p* to a state s_2. The total number of binary variables in the structure will be equal to the product of number of binary variable in each state by the number of states of the structure. The determination of the part that contributes to the falsification of property, like looking for all causes of non verification of property, that is to say, the variables that become critical in changing the context. We use an approximation algorithm proposed by Beer et al. [2] to determine from a counterexample the leading causes of non verification of property.

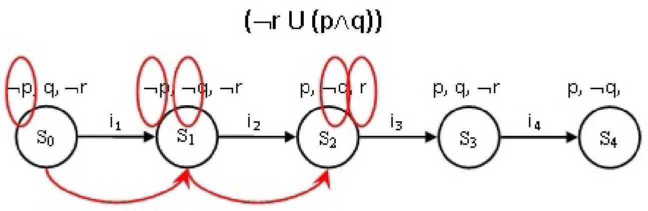


Fig. 4. Example of explanation of traces

Fig.4 shows example of explanation of traces by using the algorithm of Beer. The trace consists of 5 states *π[1..5]*. We explain counter example of the formula

$(\neg rU(p\wedge q))$. Set of causes of the failure of $\neg r$ in $\pi[1..5]$ is empty, and set of causes of the failure of $p\wedge q$ in $\pi[1..5]$ is not empty. The explanation of the formula will be $C(\pi[1..5],(p\wedge q))\cup C(\pi[2..5],(\neg rU(p\wedge q)))$ We proceed in the same way, an explanation of $(\neg rU(p\wedge q))$ in $\pi[2..5]$ is $C(\pi[2..5],(p\wedge q))\cup C(\pi[3..5],(\neg rU(p\wedge q)))$, for sub trace $\pi[3..5]$, the two sets of causes of the failure of $\neg r$ and of $p\wedge q$ is not empty, $C(\pi[3..5],(\neg rU(p\wedge q)))$ will be $C(\pi[3..5],(p\wedge q))\cup C(\pi[3..5],\neg r)$, so the set of causes of the failure of $(\neg rU(p\wedge q))$ in $\pi[1..5]$ is $\{(S_0,p),(S_1,p),(S_1,q),(S_2,q),(S_2,r)\}$.

In the case of compound formulas, the explanation is done by composition of basic explanation operators. It starts first by putting the formula in a negation normal form then we apply the explanation rules of the basic operators, and by composition find the explanation of the formula. For example, let us consider the formula $\neg(G(p)\wedge(\neg rUq))$. We start by pushing the negation, we obtain the following form $F(\neg p)\vee G(r)\vee(\neg q\wedge(X(\neg q)Ur))$ Then we compose the explanation of basic operators and we obtain $C(\pi[i..k],F(\neg p)\vee G(r)\vee(\neg q\wedge(X(\neg q)Ur)))$ which is equal to $C(\pi[i..k],F(\neg p))\cup C(\pi[i..k],G(r))\cup C(\pi[i..k],\neg q\wedge(X(\neg q)Ur))$.

Given a counterexample $\pi[0..k]$ of a formula $F(p)$, such as p is an atomic formula. The explanation of the error will be composed by the entire track (according to the Beer algorithm), thus the fragmentation of the operator G selects all states of the trace. This may not be effective in some cases (when the process is bigger and complex). The problem arises in the definition of fragment itself. The question that arises is how to measure the quality of fragmentation? We will see later the definition of a fragment from two different points of view.

The objective of Beer et al. was the determination of the leading causes of non verification of a property. They explain the failure of a type formula $G\ (p)$ by the first state in which p is not verified and not search other states. In terms of fragmentation, the fragment will be built only from the first state, the other possible states are ignored. This result may be acceptable, but it will be interesting to capture most possible states which contribute to $\neg p$.

4.5 Construction of fragments

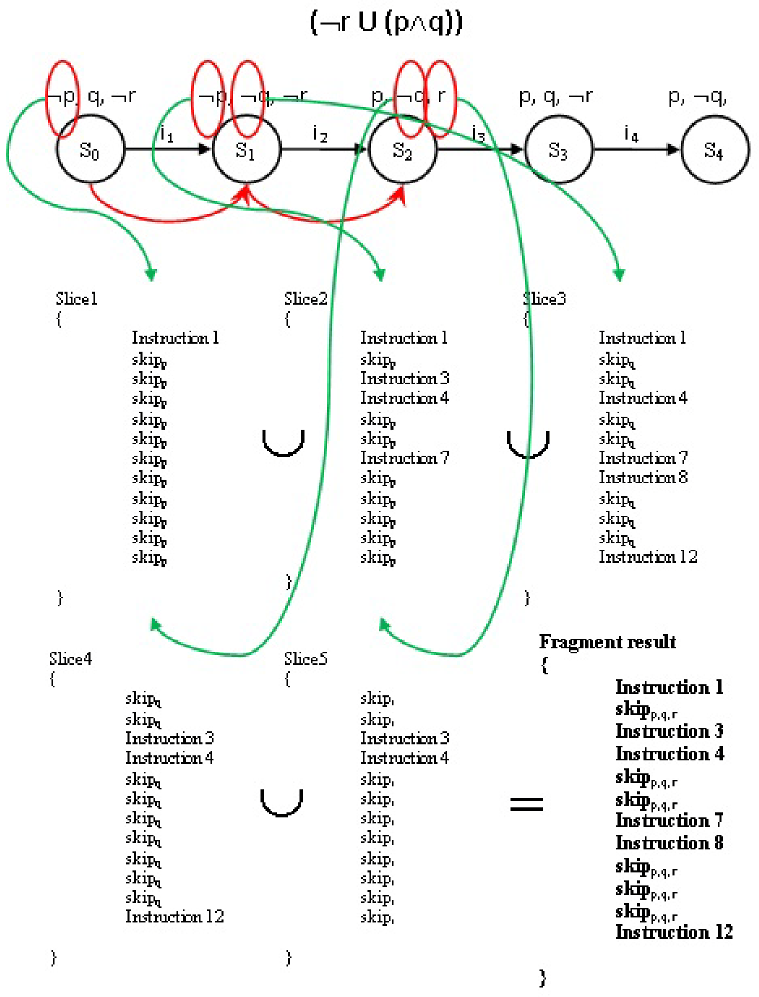
The last phase of the process of fragmentation is the construction phase of the fragment. The objective of this step is to find statements that contribute to changes impacting the state of variables involved in the explanation of the trace (previous step). For this, we use techniques of slicing concurrent programs, Nanda et al. [30].

A fragment will include a portion of the fragmented process statements. As temporal logic is based on the concept of time, if we eliminate a set of statements, we will eliminate a set of states, that is to say we will adjust the chronology. This may affect the validity of the property on a slicing particularly in the case of a formula with a "Next" operator. To solve this problem we replace each eliminated statement by a statement that does nothing *(skip)*. The fragment has the same structure as the fragmented process. The correct erasing of such skip is still possible for stuttering LTL formula.

Any statement involved in the change of state variables is considered as being part of the fragment. The idea is to seek any statements with which the selected variables in explanation step depend. We will perform a slicing for all causes selected by the previous step, and then we construct the fragment by the union of all found program slice. A slicing requires two inputs: a variable and a point in the program. In our work, a cause will be considered as an input slicing knowing that:

— The slicing variable is the boolean variable of cause.
— The slicing criterion is the transition that precedes the state of the cause.

Linear structure transitions represent program statements.



```
Slice1
{
        Instruction 1
        skip_p
        skip_p
        skip_p
        skip_p
        skip_p
        skip_p
        skip_p
        skip_p
        skip_p
        skip_p
        skip_p
}
```

```
Slice2
{
        Instruction 1
        skip_p
        Instruction 3
        Instruction 4
        skip_p
        skip_p
        Instruction 7
        skip_p
        skip_p
        skip_p
        skip_p
        skip_p
}
```

```
Slice3
{
        Instruction 1
        skip_q
        skip_q
        Instruction 4
        skip_q
        skip_q
        Instruction 7
        Instruction 8
        skip_q
        skip_q
        skip_q
        Instruction 12
}
```

```
Slice4
{
        skip_q
        skip_q
        Instruction 3
        Instruction 4
        skip_q
        skip_q
        skip_q
        skip_q
        skip_q
        skip_q
        skip_q
        Instruction 12
}
```

```
Slice5
{
        skip_r
        skip_r
        Instruction 3
        Instruction 4
        skip_r
        skip_r
        skip_r
        skip_r
        skip_r
        skip_r
        skip_r
        skip_r
}
```

```
Fragment result
{
        Instruction 1
        skip_p,q,r
        Instruction 3
        Instruction 4
        skip_p,q,r
        skip_p,q,r
        Instruction 7
        Instruction 8
        skip_p,q,r
        skip_p,q,r
        skip_p,q,r
        Instruction 12
}
```

Fig. 5. Construction of a fragment

Fig.5 reuses the example of fig.4. It summarizes the construction phase of the fragment. We perform a slicing for each case determined by explanation step, and then we will make the union of all the resulting slice programs to build the fragment.

In the slicing of a variable *x* we replace each statement eliminated by $skip_x$ which can be any statement that does not change the value of *x*. We define the intersection of two program slices by another program slice constructed by the intersection line by lines of the two program slices such as $skip_x \cup skip_y = skip_{x,y}$ and $skip_x \cup statement = statement$.

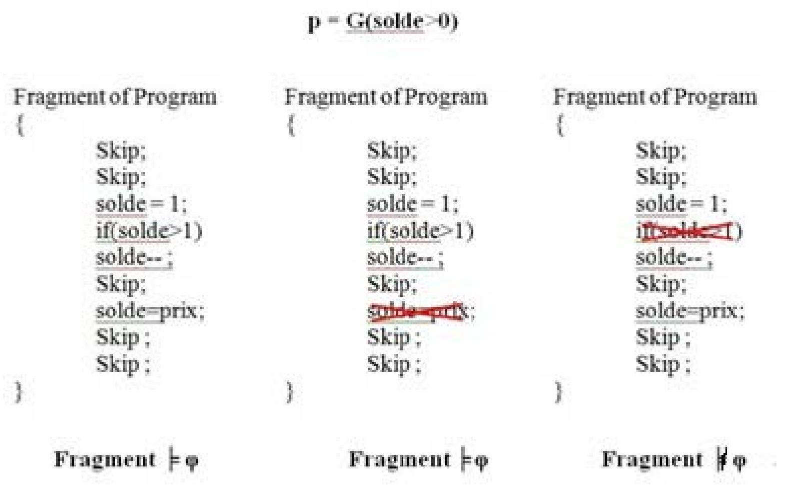
4.5.1 Refinement of a fragment

Let us consider an example of a process of online purchase. We assume that the bank would not accept a negative account, so for a purchase transaction to be validated, the client must have an account greater than or equal to the product price. We can say that throughout the buying process the customer's account is greater than or equal to zero, therefore the property *(account >= 0)* is checked every time, therefore *p = G(account >= 0)* is valid. Looking now the fragmentation according *p*. It must exist in the process a test "*if(account >= price) account = account* – price *;"*. It is clear that if we remove the test "*if(account >= price)*", the property could be violated. This test is necessary for the preservation of property *(account >= 0).* Returning to our fragmentation, how to define the fragment? Is it just that the test constitute a fragment since it is the only statement necessary for the preservation of property? From another point of view, we can say that the other statements also contribute to verification of the property because they do not violate the property. The problem is that a formula of "*G*" is checked through the whole process, so is it possible to isolate in the case of the operator "*G*" a part of the process? To try to answer this question, we propose an extension of our approach based on another definition of a fragment.

Because of these characteristics, the treatment of the "*G*" operator is special. The question is philosophical: how to define a fragment based on rules verified in all statements of a process?

We define a fragment as the necessary part which contributes to the verification and preservation of property that is the part such that if one removes the property it will not be verified.

Fig.6 illustrates this definition. We see that in the first case the property remains true despite the elimination of a statement but it becomes false in the second case, therefore it is clear that there are differences between the statements of a fragment. To study this, we proposed a refinement of the fragment to try to produce more relevant fragments. The question is then what are the criteria of the quality of a fragment?



p = G(solde>0)

Fragment of Program
{
Skip;
Skip;
solde = 1;
if(solde>1)
solde-- ;
Skip;
solde=prix;
Skip ;
Skip ;
}

Fragment of Program
{
Skip;
Skip;
solde = 1;
if(solde>1)
solde-- ;
Skip;
~~solde=prix~~;
Skip ;
Skip ;
}

Fragment of Program
{
Skip;
Skip;
solde = 1;
i~~f(solde>1~~)
solde-- ;
Skip;
solde=prix;
Skip ;
Skip ;
}

Fragment $\models \varphi$ Fragment $\models \varphi$ Fragment $\not\models \varphi$

Fig. 6. Difference between the statements of a fragment

Consider a program P, P satisfies a LTL formula Q. We say that a subset of instructions H of P is necessary for verification of Q, if the elimination of this set will result the falsification of Q. The set H is minimal if there does not exist a subset which is necessary for the verification of Q. Our goal is to determine all sets of minimal set of necessary instructions for the verification of Q (the fragment may be built by the union of these sets). It is obvious that this property (non-existence of a minimum set of necessary instructions for the verification of Q) is an anti-monotony property, because it is clear that, if a set of instructions violates the constraint, all sets constructed from him violate also the constraint.

The refinement of a fragment is based on the idea of the apriori algorithm [1]. We construct a lattice from combinations of statements of fragment (without using the *skip*), the objective is to seek the smallest combinations of statements needed by browsing through the lattice level, if a necessary node is found then all combinations including the node will be eliminated. The idea is to minimize the size of the fragment, to minimize his complexity.

The notion of refinement of a fragment introduced may appear similar to the notion of p-slice introduced in [8]. However a P-slice would only highlight relevant instruction regarding changes about predicate. This is a kind of abstraction. Our slice would rather explain which instructions contribute to a given property. Further investigations are needed to compare the technics.

5 Discussion

In our work, traces are finite. In some systems the use of infinite traces is necessary. For example, a fragmentation regarding a formula that includes a "*G*" operator in a system without a final state requires infinite traces. The extension of our approach will be necessary for this type of systems.

In an explanation step we used the algorithm of Beer et al. This algorithm provides an approximate set of causes and does not provide additional information on the causes, all provided causes are not necessarily minimum (for example in figure 4 the binary variable *p* in state s_0 is critical, the inversion of this variable is sufficient for the formula to become true in the trace). Searching all possible causes is NP-complete. It is then necessary to improve this step because the quality of fragmentation depends directly on the quality of the explanation.

In the refinement of the fragment, our goal is to focus on the difference between two types of statements in a fragment, statements necessary for the preservation of property and statements that are not required. The problem of finding the combination of statements that we presented is NP-complete. It is obvious that if the fragment is large and pruning is low, the algorithm will not perform. Our main objective is to raise the issue. We want to see the benefits brought by a fragment if we change the program. It will be interesting to investigate this issue.

6 Conclusion and perspective

In this paper, we investigated how model checking and slicing techniques can be combined to perform fragmentation of a web service composition.

We start by proving that the negation of a business rule is not checked, by generating a counterexample using a model checker. We modeled a counterexample as a linear time structure, and then we search the states of the structure responsible for the failure. From these statements we determine the set of statements that form the fragment. The advantage of our approach is the possibility of fragmenting according to temporal properties, the fragment is oriented business but it remains to validate the approach by testing to discover the performance of the method and feasibility in terms of complexity.

One lack of our approach is the ad-hoc combination of techniques to tackle the problem of fragmentation. A uniform theory is still needed and temporal theorem proving seems to be a good candidate.

In our work, we fragmented business processes according to business rules specified in LTL. It will be interesting to extend the fragmentation rules expressible in branching temporal logic (CTL). It is also interesting to extend our model to work with infinite traces which cannot be avoided in some systems for a fragmentation regarding the "*G*" operator. Among our future work, implementation of the approach to test and study the feasibility and performance of the model on real applications. We intend to design and implement a framework which does not depend on the format of the definition of web service composition, whether a BPEL code or other things. We also plan to improve the refinement of a fragment to explore this issue and investigate whether the refinement of a fragment is necessary.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In Proceedings of the 20th International Conference on Very Large Data Bases, pages 487-499. Morgan Kaufmann Publishers Inc., 1994.
2. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining Counterexamples Using Causality. In Computer Aided Verification, pages 94-108. 2009.
3. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes with BP-QL. In VLDB, pages 1255-1258, 2005.
4. The Business Rules Group, Defining Business Rules: What Are They Really ?, July 2000, http://www.businessrulesgroup.org/first_paper/br01c0.htm.
5. H. Chockler, J.Y. Halpern, and O. Kupferman. What causes a system to satisfy a specification? ACM Transactions on Computational Logic (TOCL), 9(3):20, 2008.
6. E.M. Clarke, E.A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. Communications of the ACM, 52(11):74-84, 2009.
7. E. Clarke, O. Grumberg, and D. Peled D. Model Checking. MIT Press, 1999.
8. J. J. Comuzzi, J. M. Hart. Program Slicing Using Weakest Preconditions. FME, pages 557-575, 1996.
9. O. Danylevych, D. Karastoyanova, and F. Leymann. Optimal stratification of transactions. In 2009 Fourth International Conference on Internet and Web Applications and Services, pages 493-498. IEEE, 2009.
10. C. Dixon, M. Fisher, and H. Barringer. A graph-based approach to resolution in temporal logic. In ICTL, pages 415-429, 1994.
11. C. Dixon, M. Fisher, and B. Konev. Is There a Future for Deductive Temporal Verification? In Temporal Representation and Reasoning, 2006. TIME 2006. Thirteenth International Symposium on, pages 11-18, 2006.
12. T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. In IJCAI, pages 35-42, 2001.
13. A. Emerson. Temporal and Modal Logic. Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B): 995-1072, MIT Press, 1991.
14. J. Arias-Fisteus, L.S. Fernández, and C.D. Kloos. Applying model checking to BPEL4WS business collaborations. In Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), ACM, pages 826-830, 2005.
15. M. Fisher. A resolution method for temporal logic. In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI), pages 99-104. Citeseer, 1991.
16. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In Proceedings of the 13th international conference on World Wide Web, pages 621-630. ACM, 2004.
17. D. M. Gabbay and A. Pnueli. A sound and complete deductive system for ctl* verification. Logic Journal of the IGPL, 16(6):499-536, 2008.
18. J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach - Part I: Causes. In UAI, pages 194-202, 2001.
19. J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach - Part II: Explanations. In IJCAI, pages 27-34, 2001.
20. G.J. Holzmann. The SPIN model checker: Primer and reference manual. Addison Wesley Publishing Company, 2004.
21. U. Hustadt, B. Konev, A. Riazanov, and A. Voronkov. Temp: A temporal monodic prover. In In Proc. IJCAR-04, LNAI, pages 326-330. Springer, 2004.

22. R. Ivanovic, M. Carro, M. Hermenegildo. Sharing-Based Independence-Driven Fragment Identification for Service Orchestration, Network of Excellence S-Cube, 2010.
23. R. Kazhamiakin, M. Pistore, and M. Roveri. Formal verification of requirements using spin: A case study on web services. In SEFM, pages 406-415, 2004.
24. R. Khalaf. Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective. Doctoral thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, March 2008.
25. A. Lomuscio, H. Qu, M. Sergot, and M. Solanki. Verifying temporal and epistemic properties of web service compositions. Service-Oriented Computing-ICSOC 2007, pages 456-461.
26. Z. Manna and A. Pnueli. Completing the temporal picture. Theor. Comput. Sci., 83(1):97-130, 1991.
27. Z. Manna and A. Pnueli. Temporal verification of reactive systems: safety. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
28. K. L. McMillan. Interpolation and SAT-Based Model Checking. In Computer Aided Verification, pages 1-13. 2003.
29. Z. Milosevic, S. W. Sadiq, and M. E. Orlowska. Towards a methodology for deriving contract-compliant business processes. In Business Process Management, pages 395-400, 2006.
30. M.G. Nanda and S. Ramesh. Slicing concurrent programs. In Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, page 190. ACM, 2000.
31. D. Neiger and L. Churilov. Goal-oriented business process modeling with EPCs and value focused thinking. Business Process Management, pages 98-115, 2004.
32. M. Papazoglou. Web Services: Principles and Technology. Pearson - Prentice Hall, 782 pages, 2007.
33. R. G. Ross. Expressing business rules. In SIGMOD Conference, pages 515-516, 2000.
34. Algorithms and Techniques for Splitting and Merging Service Compositions. Deliverable PO-JRA-2.2.3, S-Cube Consortium, November 2009. http://www.s-cube-network.eu/results/deliverables/wp-jra-2.2, 2009.
35. Network of Excellence S-Cube. http://www.s-cube-network.eu/, 2008.
36. SPIN Model Checker. http://spinroot.com/spin/whatispin.html.
37. F. Tip. A survey of program slicing techniques. Journal of programming languages, 3(3):121-189, 1995.
38. J. Vanhatalo, H. Volzer, and F. Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. Service-Oriented Computing-ICSOC 2007, pages 43-55, 2007.
39. WSAT: Web Service Analysis Tool. http://www.cs.ucsb.edu/~su/WSAT/.
40. S. Zaplata, K. Kottke, M. Meiners, and W. Lamersdorf. Towards Runtime Migration of WS-BPEL Processes. WESOA 2009.
41. S. Zaplata, K. Hamann, and W. Lamersdorf. Executing Parallel Tasks in Distributed Mobile Processes. Network of Excellence S-Cube, 2010.
42. X. Zhao, H. Yang, and Z. Qiu. Towards the formal model and verification of web service choreography description language. Proc. of WS-FM 2006, pages 273-287, 2006.