The Journal of Systems and Software 89 (2014) 109–127

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A reliability model for *Service Component Architectures*

R. Mirandola [c], P. Potena [b], E. Riccobene [a,*], P. Scandurra [b]

[a] Computer Science Department, Univ. degli Studi di Milano, Crema, CR, Italy
[b] Engineering Department, Univ. degli Studi di Bergamo, Dalmine, BG, Italy
[c] Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy

ARTICLE INFO

Article history:
Received 19 October 2012
Received in revised form 21 June 2013
Accepted 3 November 2013
Available online 27 November 2013

Keywords:
Software reliability models
Service Component Architecture
Abstract State Machines

ABSTRACT

Service-oriented applications are dynamically built by assembling existing, loosely coupled, distributed, and heterogeneous services. Predicting their reliability is very important to appropriately drive the selection and assembly of services, to evaluate design feasibility, to compare design alternatives, to identify potential failure areas and to maintain an acceptable reliability level under environmental extremes.

This article presents a model for predicting reliability of a service-oriented application based on its architecture specification in the lightweight formal language SCA-ASM. The SCA-ASM component model is based on the OASIS standard *Service Component Architecture* for heterogeneous service assembly and on the formal method *Abstract State Machines* for modeling service behavior, interactions, and orchestration in an abstract but executable way.

The proposed method provides an automatic and compositional means for predicting reliability both at system-level and component-level by combining a reliability model for an SCA assembly involving SCA-ASM components, and a reliability model of an SCA-ASM component. It exploits ideas from *architecture-based* and *path-based* reliability models. A set of experimental results shows the effectiveness of the proposed approach and its comparison with a state-of-the art BPEL-based approach.

1. Introduction

In service-oriented computing, software applications are dynamically built by assembling existing, loosely coupled, distributed, and heterogeneous services. It has been widely recognized (Smith and Williams, 2002; Cardellini et al., 2012; Calinescu et al., 2012) that the prediction of non-functional properties of these systems is a crucial design-time concern. Architectural decisions, indeed, including selection of services and the structure of the workflow, may significantly affect the qualities of the resulting system, such as their reliability, performance, or cost. This paper focuses on *reliability*. Early assessment of reliability is one of the challenges of service-oriented architectures and a key factor to developing dependable software.

Service-oriented architectures allow designers to reason on systems' reliability at a higher level of abstraction. At the modeling level, services can be viewed as black-box units. Based on a model of the architecture, which describes how services are connected together and interact, we expect well-founded methods to be available to software engineers to reason about satisfaction of the global system's reliability requirements.

In the literature, different procedures exist for system reliability prediction based on different assumptions and applicable at different granularity of information (Immonen and Niemelä, 2008; Grassi, 2004; Goseva-Popstojanova and Trivedi, 2001; Filieri et al., 2010; Cardellini et al., 2012). These techniques can be applied for several purposes such as: to evaluate design feasibility, to compare design alternatives, to assist in evaluating the significance of reported failures, to trade-off system design factors, to track reliability improvement, to appropriately allocate validation/testing effort, and to identify potential failure areas and maintain an acceptable reliability level under environmental extremes.

In this paper, we propose a reliability prediction method for the SCA-ASM component model (Riccobene et al., 2011; Riccobene and Scandurra, 2013), which is automatic and compositional. The SCA-ASM has been recently proposed as a lightweight formal language for modeling both architecture and behavior aspects of a service application. This component model is based on the OASIS open standard *Service Component Architecture* (SCA) for heterogeneous service assembly, and on the formal method *Abstract State Machines* (ASMs) (Börger and Stärk, 2003) for modeling notions of service behavior, interactions, orchestration, fault and compensation handling in an abstract but executable way. Since the SCA-ASM

* Corresponding author. Tel.: +39 0250330055.
E-mail addresses: raffaela.mirandola@polimi.it (R. Mirandola), pasqualina.potena@unibg.it (P. Potena), elvinia.riccobene@unimi.it (E. Riccobene), patrizia.scandurra@unibg.it (P. Scandurra).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.11.002

relies on the SCA design framework, it is supported by the runtime environment Tuscany, thus simplifying the prototyping, analysis, development, and deployment of service compositions.

The reliability model we propose here exploits ideas taken from *architecture-based* and *path-based* reliability models (Goseva-Popstojanova and Trivedi, 2001). It is an extension of the model presented in Riccobene et al. (2012) where the reliability model of an SCA-ASM component was left abstract. We present a reliability model for an SCA assembly involving SCA-ASM components that embed the main service orchestration. We also introduce a reliability model of an SCA-ASM component by considering failures specific to the nature of the ASMs.

Besides to be compositional and applicable at design phase as well as at run-time, there are some other potential advantages of our approach. We rely on a unique component model, i.e. SCA-ASM, that is both the "design-oriented model" of the component assembly and the "formal analysis-oriented model" that leads the reliability analysis. Instead, there are many approaches developed over the past ten years that directly tie architectural models (or flavors of UML and other modeling notations) to formal reliability models such as Markov or Bayesian models (see related works in Section 7). W.r.t. these classical approaches, a novelty aspect here is also the combination of the reliability prediction of the service orchestrator with those of other service components; this leads to a more accurate estimation of the reliability. An experimental analysis illustrates the usage and the effectiveness of the proposed approach. Moreover, for the sake of comparison, we have conducted a parametric numerical evaluation of the reliability obtained with the SCA-ASM model proposed in this paper and with one of the state-of-the art models considering a BPEL-based service composition.

This paper is organized as follows. Section 2 recall some basic concepts concerning reliability prediction, and provides some background on the SCA-ASM formal modeling language both for the architectural description and for the behavioral specification. Sections 3 and 4 describe, respectively, the reliability model for an SCA assembly and for an SCA-ASM component. Two kinds of behavioral failure are taken in to consideration. Section 5 provides a case study inspired by the "Finance case study" of the EU project SENSORIA. Section 6 presents the results of the parametric numerical evaluation. Section 7 describes some related work. Finally, Section 8 concludes the paper and sketches some future directions.

2. Background concepts

This section provides those background concepts useful to understand the reliability prediction method we here propose for service-oriented applications. We introduce some basic concepts on reliability prediction, and we present the SCA-ASM lightweight formal language adopted for modeling the Service Component Architecture. The proposed reliability prediction method combines a reliability model for an SCA assembly of SCA-ASM components, and a reliability model of an SCA-ASM component.

2.1. Reliability prediction basics

Reliability is one of the major factors of software quality and is defined as the "probability of failure-free software operation for a specified period of time in a specified environment" (Standard, 1991). *Reliability prediction* is a common form of reliability analysis to predict the failure rate of components and the overall system reliability. Reliability predictions are useful to evaluate design feasibility, compare design alternatives, identify potential failure areas, trade-off system design factors, and track reliability improvement. A reliability prediction can also assist in evaluating the significance of reported failures and it can be used to maintain an acceptable reliability level under environmental extremes. Reliability strongly depends on two main concerns. First, the reliability of a software system depends on the reliability of individual components, component interactions, and the execution environment. Second, reliability depends on how the system will be used (*usage profile* or *operational profile*). Since reliability (like availability) is an execution quality, the impact of faults on reliability differs depending on how the system is used, i.e. how often the faulty part of the system is executed. The analysis of different ways and frequencies to execute the system is a challenge to reliability prediction, especially when the usage profiles are unknown beforehand.

In the last few years many reliability prediction methods for software have been introduced (Filieri et al., 2010; Immonen and Niemelä, 2008; Goseva-Popstojanova and Trivedi, 2001). Basically, the existing techniques can be classified as *path-based models* and *state-based models* (Goseva-Popstojanova and Trivedi, 2001). The former ones represent the system architecture as a combination of the possible execution paths, whereas the latter ones as a combination of the possible states of the system.

In the following we describe the main assumptions underlying our reliability model. Most of them are common to many existing reliability approaches (see, for example, the surveys Krka et al., 2009; Immonen and Niemelä, 2008) and are necessary to be able to provide in an efficient way analytical results that, even if approximate with respect to the more complex reality, can give meaningful insights to system designers. Our main assumptions are:

(i) The components communicate by exchanging synchronous messages.
(ii) The components' failures are independent of each other. We assume that a component's failure provokes the crash of the whole system, namely the system straightforwardly stops its execution. The inclusion in our model of different types of failures and of error propagation analysis is at present under study.
(iii) Model parameters' uncertainties (Chandran et al., 2010) are not dealt. It was out of the scope of this paper to deal with this kind of sensitivity analysis. We consider this to be an interesting avenue of future research.

2.2. Service Component Architecture (SCA)

SCA is an XML-based component model used to develop service-oriented applications independently from SOA platforms and middleware programming APIs (like Java, C++, Spring, PHP, BPEL, Web services, etc.). SCA is also supported by a visual notation (a metamodel-based language developed with the Eclipse-EMF) and runtime environments (like Apache Tuscany, FRAscaTI, IBM WebSphere Application Server V7, to name a few) to create service components, assemble them into a composite application, provide an implementation for them, and then run/debug the resulting composite application.

According to the principles of service-oriented computing, loosely coupled service components are used as atomic units or building blocks to build an application. Fig. 1 shows an *SCA composite* (or *assembly*) as a collection of SCA components. An *SCA component* is a piece of software that has been configured to provide its business functions (operations) for interaction with the outside world. This interaction is accomplished through: *services* that are externally visible functions provided by the component; *references* (functions required by the component) wired to services provided by other components; *properties* allowing for the configuration of a component implementation and *bindings* that specify access mechanisms used by services and references according to some technology/protocol (e.g. WSDL binding to consume/expose

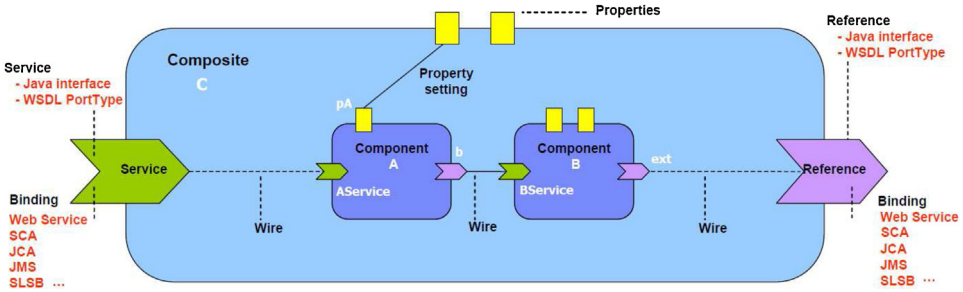

Fig. 1. An SCA composite (adapted from the SCA Assembly Model V1.00 spec.)

web services, JMS binding to receive/send Java Message Service, etc.). Services and references are typed by *interfaces*. An interface describes a set of related operations (or business functions) which as a whole make up the service offered or required by a component. The provider may respond to the requester client of an operation invocation with zero or more messages. Message exchange may be synchronous or asynchronous. As unit of composition and hierarchical design, assemblies of components deployed together are called *composite* components and consist of: properties, services, services organized as sub-components, required services as references, and wires connecting sub-components. A top level composite describes the overall assembly (service-oriented application).

2.3. The SCA-ASM component model

This section sketches some basic notions concerning the *Abstract State Machines (ASM)* formal method (Börger and Stärk, 2003) and the *SCA-ASM modeling language* (Riccobene et al., 2011; Riccobene and Scandurra, 2013). SCA-ASM complements the SCA component model with the "model of computation" of the ASM formalism. The aim is to define a new SCA *component implementation type* to provide ASM-based formal stateful and executable descriptions of the services *internal behavior*, *orchestration* and *interactions*. An SCA-ASM design environment (Brugali et al., 2011) was developed by integrating the Eclipse-based SCA Composite Designer, the SCA runtime platform Tuscany, and the simulator AsmetaS (Arcaini et al., 2011) of the ASM specification and analysis toolset ASMETA (ASMETA, 2011).

ASMs are an extension of Finite State Machines (FSMs) (Börger and Stärk, 2003) where unstructured control states are replaced by states of arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (Boolean functions) defined on them. The *transition relation* is specified by rules describing how functions change from one state to the next. There is a concise but powerful set of ASM *rule constructors*, but the basic transition rule has the form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed[1] when *Condition* is true. Functions changing as a consequence of *updates* are *dynamic* and they are further classified in: *monitored* (only read, as events provided by the environment), *controlled* (read and written by the machine), *shared* (read and written by the machine and by the environment) and *out* (only written by the machine) functions.

According to the SCA-ASM component implementation type, a service-oriented component is an ASM endowed with (at least) one agent (a business partner or role) able to interact with other agents by providing and requiring services to/from other service-oriented components' agents. The service behaviors encapsulated in an SCA-ASM component are captured by ASM transition rules.

Fig. 2 shows the shape of an SCA-ASM component `A` and the corresponding ASM modules for the provided interface `AService` (on the left) and the skeleton of the component itself (on the right) using the textual notation AsmetaL (Gargantini et al., 2008)[2] of the toolset ASMETA. The `@annotations` are used to denote SCA concepts such as references, properties, etc.

The ASM module for the component `A` provides definitions for the business functions declared in the imported ASM module `AService` corresponding to the provided interface `AService`. The module `A` also provides declarations for the property `pA`, the reference `b` to an agent `BService`, a back reference `client` to the requester agent, and other functions. An ASM agent is associated to the SCA-ASM and executes a specific *program* (a named ASM transition rule) as its behavior. To this purpose, an SCA-ASM component has a distinguished rule name of arity zero taking by convention the same name of the component. The agent domain `AService`, for example, declared in the interface module `AService` in Fig. 2 characterizes the agent associated to the component `A`. The rule `r_A` defined in the module `A` is assigned as program to the `A` component's agent. This agent is created during the component initialization and its program is used as entry point for the component execution.

The formal definition of SCA-ASM component can be found in Riccobene and Scandurra (2013). A working definition follows.

Definition 1. An *SCA-ASM component* is an ASM module of form

(*header*, *body*)

The *header* is the tuple

(*name*, *prov_services*, *req_services*, *signature*, *import*, *export*)

and the *body* is the tuple

(*doms_and_functs*, *invs*, *rules*, *services*, *prog*, *init*, *handlers*)

where

name is the component name;
prov_services and *req_services* are import clauses annotated, respectively, with `@Provided` and `@Required`, to include the ASM modules of the service interfaces provided/required by the component;
signature is defined as the tuple (*pro_decl*, *ref_decl*, *dom_and_funct_decl*) and contains declarations for externally

[1] f is an n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. To fire this rule in a state S_i, $i \geq 0$, evaluate all terms $t_1, \ldots, t_n, t$ at S_i and update the function f to t on parameters $t_1, \ldots, t_n$. The next state S_{i+1} differs from S_i only in the new interpretation of f.

[2] Two grammatical conventions must be recalled: a variable identifier starts with $; a rule identifier begins with "*r_*".

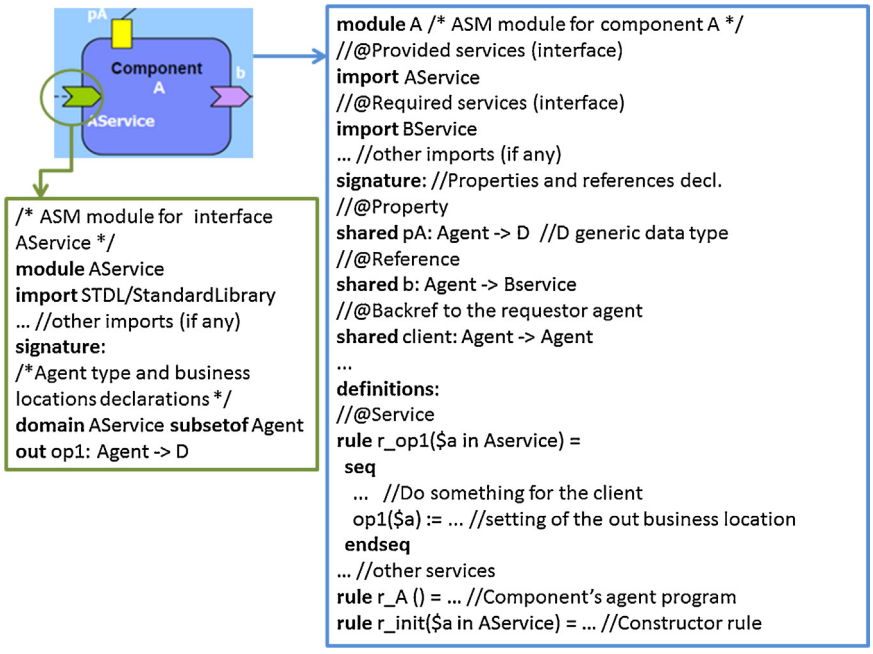

Fig. 2. SCA-ASM component shape.

settable *property* values (i.e., ASM monitored functions – or shared functions when promoted as a composite property – annotated with `@Property`), declarations for *references* (ASM controlled functions annotated with `@Reference`), and declarations of other ASM *domains* and *functions* to be used by the component for internal computation only. In SCA-ASM, references are represented as functions (annotated with `@Reference`) having as codomain a subset of the `Agent` domain named with the name of the reference's typing interface (see, e.g., the reference `b` to a `BService` agent in the ASM module `A` in Fig. 1). This domain is declared in the ASM module corresponding to the reference's typing interface; the ASM module corresponding to the component exposing the interface has also to import the ASM module for the interface. Thus, we identify (even if it is not known at design time) the partner's business role (i.e., the agent type). Back references to requester agents are modeled as functions in the same way (using the annotation `@Backref`), but the agent codomain is the most generic one, i.e., the `Agent` domain.

import and *export* specify other module libraries that are included;

doms_and_functs are definitions of domains and functions (static concrete-domains and static/derived functions) already declared in the signature;

invs are definitions of state invariants (eg., first-order formulas over some functions of the ASM which must hold in every state of the ASM);

rules are definitions of (utility) transition rules for internal computation;

services are definitions of services (i.e., definition of transition rules annotated with `@Service`);

prog is the definition of a *main transition rule* (that takes by convention the same name of the component's module) to assign as "program" to the component's agent created during the component initialization;

init is the definition of the transition rule with the predefined name `r_init` that is in turn invoked during initialization to set the internal state (controlled functions) of the SCA-ASM component;

handlers are definitions of transition rules, annotated with `@ExceptionHandler` and `@CompensationHandler`, fired as, respectively, exception and compensation handlers in case of faults.

In addition to the basic function update rule, ASM rule constructors and predefined ASM rules (i.e., named ASM rules made available as model library) are used as SCA-ASM behavioral primitives. They are summarized in Table 1 by separating them according to the separation of concerns *computation* and *coordination*, and *communication*.[3] In particular, communication primitives provide both synchronous and asynchronous interaction styles (corresponding, respectively, to the *request-response* and *one-way* interaction patterns of the SCA standard). Communication relies on an *abstract message-passing* mechanism by adopting the default SCA binding (`binding.sca`) for message delivering. SCA-ASM rule constructors can be combined to model specific interaction and orchestration patterns in well structured and modularized entities. SCA-ASM modeling constructs for fault/compensation handling are also supported (see Riccobene et al., 2011), but are not reported here since we do not take into account fault tolerance concepts in the reliability model proposed here.

Currently, the implementation scope of an SCA-ASM component is *composite*, i.e. a single component instance (a single ASM) is created for all service calls of the component. The other two SCA implementation scopes, *stateless* (to create a new component instance for each service call) and *conversation* (to create a component instance for each conversation), are not yet supported.

Below, two definitions follow about the computational semantics of an SCA-ASM component (or SCA-ASM machine) within an SCA assembly, referred by us as *in-place simulation* semantics (Brugali et al., 2011).

Definition 2. A mov_i from a machine state s_{i-1} to the state s_i is a single computation step executed by an SCA-ASM component (or SCA-ASM machine). It consists into firing the updates produced by the program of the machine, if they do not clash. Since the main rule of an SCA-ASM does not have parameters and there are no free global variables in the rule declarations of an SCA-ASM, the notion of a move does not depend on a variable assignment.

[3] The formal semantics of such rules is given in Riccobene and Scandurra (2013). Note that, differently from Riccobene and Scandurra (2013), here the names of the communication primitives start with the prefix "w".

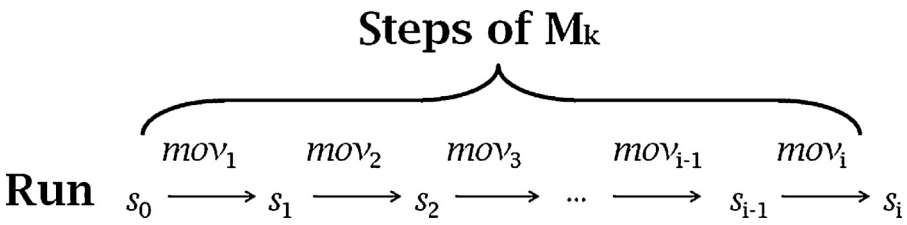

Fig. 3. The SCA-ASM notions of move and run.

Table 1
SCA-ASM rule constructors.

Computation and Coordination	
Skip rule	**skip** do nothing
Update rule	$f(t_1, \ldots, t_n) := t$ update the value of f at $t_1, \ldots, t_n$ to t
Call rule	$R[x_1, \ldots, x_n]$ call rule R with parameters $x_1, \ldots, x_n$
Let rule	**let** $x = t$ **in** R assign the value of t to x and then execute R
Conditional rule	**if** ϕ **then** R_1 **else** R_2 **endif** if ϕ is true, then execute rule R_1, otherwise R_2
Iterate rule	**while** ϕ **do** R execute rule R until ϕ is true
Seq rule	**seq** $R_1 \ldots R_n$ **endseq** rules $R_1 \ldots R_n$ are executed in sequence without exposing intermediate updates
Parallel rule	**par** $R_1 \ldots R_n$ **endpar** rules $R_1 \ldots R_n$ are executed in parallel
Forall rule	**forall** x **with** ϕ **do** $R(x)$ forall x satisfying ϕ execute R
Choose rule	**choose** x **with** ϕ **do** $R(x)$ choose an x satisfying ϕ and then execute R
Split rule	**forall** $n \in N$ **do** $R(n)$ split N times the execution of R
Spawn rule	**spawn** child **with** R create a child agent with program R
Communication	
Send rule	**wsend**[*lnk*,*R*,*snd*] send data *snd* to *lnk* in reference to rule R (no blocking, no acknowledgment)
Receive rule	**wreceive**[*lnk*,*R*,*rcv*] receive data *rcv* from *lnk* in reference to R (blocks until data are received, no ack)
SendReceive rule	**wsendreceive**[*lnk*,*R*,*snd*,*rcv*] send data *snd* to *lnk* in reference to R waits for data *rcv* to be sent back (no ack)
Reply rule	**wreply**[*lnk*,*R*,*snd*] returns data *snd* to *lnk*, as response of R request received from *lnk* (no ack)

Definition 3. A run_k of an SCA-ASM component (or SCA-ASM machine) M_k related to the invocation of the service k is a finite or infinite sequence $s_0, s_1, \ldots s_{i-1}, s_i \ldots$ of states of the machine, where s_0 is an initial state and each s_i is obtained from s_{i-1} by firing the program of M_k.

Fig. 3 illustrates the notion of abstract computation segments $mov_1, \ldots, mov_i$ of an SCA-ASM machine M_k where each mov_i represents a single M_k-move, and of run of M_k from an initial state s_0 to a state s_i.

In dynamic situations, it is convenient to view a state of an SCA-ASM machine as a kind of memory that maps locations to values.

Definition 4. A location of a state s of an SCA-ASM machine is a pair $(f, (a_1 \ldots a_n))$, where f is an n-ary function name and $a_1 \ldots a_n$ are elements of the base set (or superuniverse) of s. The value $f(a_1, \ldots, a_n)$ is called the content of the location in s. The elements of the location are the elements of the set $\{a_1, \ldots, a_n\}$.

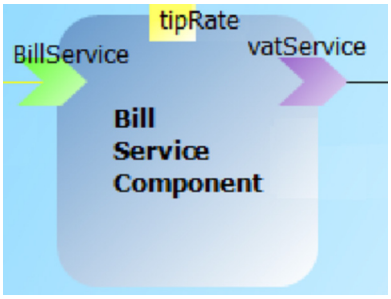

Fig. 4. An SCA BillServiceComponent.

The meaning of an update (l, v) is that the content of a location l in the state has to be changed to a value v. Since due to the parallelism a transition rule may prescribe updating the same function at the same arguments several times, we require such updates to be consistent. Two updates clash, if they refer to the same location but are distinct. Formally:

Definition 5. An update set U is *consistent*, if it contains no pair of updates with the same location, i.e., no two elements (l, v), (l, w) with $v = w$.

As long as the SCA-ASM machine can make a move, the run proceeds, requiring only that the interspersed moves of the *environment*,[4] namely updating monitored or shared functions (essentially, SCA properties of the component), produce a consistent state for the next machine move. If in a state the machine cannot produce a consistent update set or no update set at all, then the state is considered to be the last state in the run.

Running example. The Bill Service is a concise and simple service-oriented component that we use as running example in the following sections to explain definitions and concepts regarding the reliability model. Fig. 4 shows an SCA component for the bill service, slightly adapted from the `Restaurant` case study of the SCA distribution (SCA). The Bill Service computes the price of a menu[5] with the different taxes. Listings 1 shows a possible SCA-ASM implementation of such a component in AsmetaL, the textual notation of the ASM toolset ASMETA (ASMETA, 2011). An external helper service, a VAT service, is invoked in turn by the bill service to compute the final price with tax. In addition, by default there is a tip rate, the property `tipRate`, no greater than 15 (by the state invariant), but the client invoking the service can also specify a further user tip rate. If a user tip rate is specified, the default tip rate will not be considered.

[4] Read: by some other (say an unknown) agent representing the context in which the SCA-ASM machine computes, namely the SCA "container" of the component according to the underlying technology (Brugali et al., 2011).

[5] A menu is here a data type defined by a description and the price without VAT and tip taxes.

Listing 1. SCA-ASM model of a BillServiceComponent.

```
module BillServiceComponent
import STDL/StandardLibrary
import STDL/CommonBehavior
//@Provided interface
import BillService
//@Required interface
import VatService
export *
signature:
//@Reference
shared vatService: Agent → VatService
//@Backref
shared clientBillService: Agent → Agent
//@Property
shared tipRate: Agent → Real
//Other functions used for internal computation
controlled priceWithTaxRate: Agent → Real
controlled priceWithTipRate: Agent → Real
controlled menuprice: Agent → Real
controlled usrTipRate: Agent → Real
definitions:
//Invariant definitions:
invariant tipRate_inv over tipRate: tipRate > = 0.0 and tipRate ≤
  15
invariant priceWithTaxRate_inv over priceWithTaxRate:
  priceWithTaxRate > menuprice
invariant usrTipRate_inv over usrTipRate: usrTipRate ≥ 0.0 and
  usrTipRate ≤ 15
//@Service
rule r_getBill($a in Agent, $menuPrice in Real, $usrTipRate in
  Real) =
seq
  r_wsendreceive(vatService($a),"getPriceWithVat",$menuPrice,
  priceWithTaxRate($a))
  if ($usrTipRate > 0.0) //A user tip rate has been specified
  then priceWithTipRate($a):=
priceWithTaxRate($a) * $usrTipRate/100.0 +priceWithTaxRate($a)
  else priceWithTipRate($a):= priceWithTaxRate($a) *
  tipRate($a)/100.0 +priceWithTaxRate($a)
  //setting of the out business function location
  getBill($a,$menuPrice, $usrTipRate):= priceWithTipRate($a)
endseq
rule r_BillServiceComponent = //Component's program
seq
  r_wreceive(clientBillService(self),"getBill",(menuprice(self),
  usrTipRate(self)))
  r_getBill(self, menuprice(self), usrTipRate(self)) //direct
  service invocation
  r_wreply(clientBillService(self),"getBill",getBill(self,
  menuprice(self),usrTipRate(self)))
endseq
//Constructor rule
macro rule r_init($a in BillService) = usrTipRate(self):= 0.0
```

3. Reliability of an SCA assembly

In this section, we present a reliability model for an SCA assembly (an SCA application) involving SCA-ASM components that play the role of "core components" in the service orchestration. Specifically, we assume that for each service exposed by the SCA assembly (composite component) there is an SCA-ASM component, a *core component*, that provides that service on behalf of the composite by interacting with and coordinating the other SCA sub-components (possibly implemented with different programming languages). Note that this assumption is not restrictive; it is only a position with respect to the concern of coordinating (*orchestrating*) the various parties that, together, deliver a complex service. The works in Arbab (2004) and Abreu and Fiadeiro (2008), for example, adopt a "classical" architectural approach in which this type of coordination is distributed and performed by connectors that link together the different parties involved in the delivery of the service. Other approaches, like our approach, adopt workflow models (van der Aalst and Pesic, 2006). The orchestration of business roles can be much more complex. Our model, however, can be easily adapted to reflect these different levels of coordination. Note also that these core components could be implemented in a different way without essentially changing the overall model structure. In fact, the orchestration of components within composites can be realized with other workflow-oriented languages such as BPEL/WSDL, Jolie, etc. (Mayer et al., 2008).

Moreover, in order to keep the reliability model simple, we assume that such SCA-ASM core components are single-agent ASMs[6] supporting sequential and parallel computations.

At a coarse grain, failures occurring during the execution of an SCA application can be classified as follows: *crash failures*, that provoke the crash of the whole application, namely that immediately and irreversibly compromise the behavior of the whole application; *no-crash failures*, that do not provoke the immediate termination of the whole application, but manifest themselves by the return of an erroneous message. We here focus our attention on crash failures, and we assume that a failure of a component in the SCA assembly (included the SCA-ASM core components) provokes the failure of the whole application. Such assumption is not too restrictive. It is a common practice in many reliability modeling approaches (see, e.g., the survey Immonen and Niemelä, 2008). We postpone the study of no-crash failures to future work.

In the remainder of this section, we first define the SCA-ASM usage profile, then the structure of SCA-ASM Rule Dependency Tree (similar to the structure CDG in Yacoub et al., 1999), and finally we introduce our model for the reliability of an SCA assembly based on such tree-structures for the SCA-ASM core components.

3.1. SCA-ASM usage profile

For a service k provided by the SCA-ASM core component of an SCA assembly, the following definitions hold.

Definition 6. An *SCA-ASM usage profile* is a tuple

$$(pexec_k, (P(rule_1), P(rule_2), \ldots, P(rule_m)))$$

where

$pexec_k$ is the probability of execution for service k and can be given in input from the designer or can be derived from the observation of the SCA-ASM component behavior or from observations derived from systems offering the same type of services.
$P(rule_j)$ is the probability of executing $rule_j$, $1 \le j \le m$, supposing m is the number of occurrences of rule constructors in the SCA-ASM model for service k.[7] These probabilities are derived from historical data deriving from the observation of the SCA-ASM component behavior.

The definition of the SCA-ASM usage profile is then completed by considering a single mov_i. Its execution depends on how in the state s_{i-1} the environment updates the monitored functions.

Definition 7. A mov_i *usage profile* is a tuple $(P(m_1^i), P(m_2^i), \ldots, P(m_{fm}^i))$

where $P(m_j^i)$ denotes the probability that the environment updates correctly (i.e., input values do not cause inconsistent updates, invariant failure or any other unexpected faulty behavior) the monitored function m_j, $1 \le j \le fm$, at the move i, given that fm is the number of monitored functions.

[6] We therefore do not consider dynamic instantiation of sub-agents (by the use of the *spawn rule*).

[7] More precisely, m yields the number of rule constructors occurring in the sections *rules*, *services* and *prog* (see Definition 1) of the SCA-ASM component providing the service k.

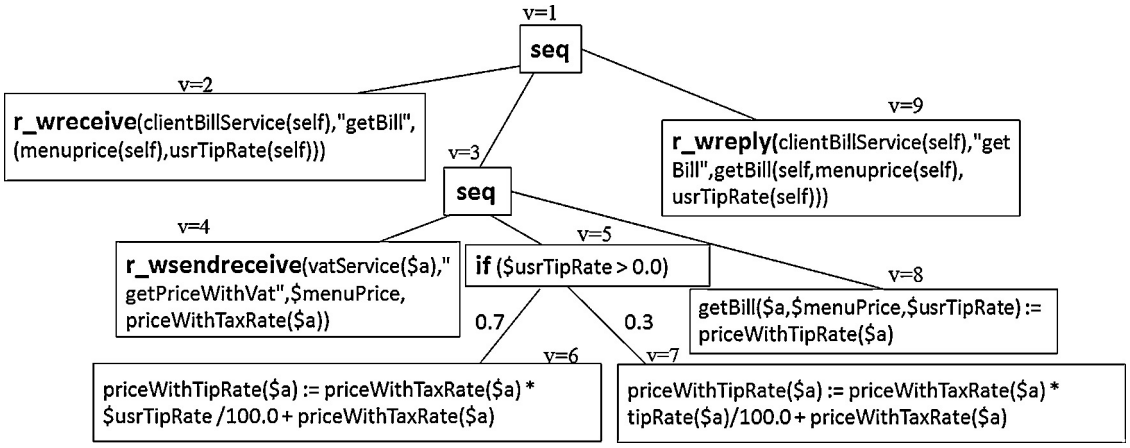

Fig. 5. RDT for the BillServiceComponent.

3.2. SCA-ASM rule dependency tree

Let C be the set of components in an SCA assembly, and K be the number of services exposed and provided by the assembly.

By the usage profile (see Definition 6), we know the probability $pexec_k$ that the kth service will be invoked and, therefore, executed. It must hold $\sum_{k=1}^{K} pexec_k = 1$.

Let us assume that for each service k exposed by the SCA assembly one of the $|C|$ components, a core component, is in charge of processing the information on the client's behalf and coordinating the other components in order to provide the service. Note that such a component may still provide other services exposed by the assembly.

Given an SCA-ASM component (or SCA-ASM machine) M, it is convenient to represent the component's agent program of M (or simply, program of M), by means of a tree that concisely captures the nesting relationship of the rules involved in its definition.

Definition 8. Given an SCA-ASM component or machine M, we define the *Rule Dependency Tree* (RDT) of M a tree structure (V, E) where a node $v \in V$ is labeled by a basic rule for computation (`skip` or update rule) or by a basic rule for communication (`wreceive` or `wsendreceive` or `wreply`) if v is a leaf node, and by a rule constructor for computation/coordination (`seq`, `par`, `if`, `forall`) if v is an internal node; the edges E reflect the direct nesting relationship among the rules of the program of M.

For the sake of model formulation, in this paper the reliability is calculated under the assumption that service invocation is *synchronous* (i.e. with a blocking effect for the caller). This means that a service operation is invoked through the communication primitive `wsendreceive`. We do not allow the use of `wsend` that has an asynchronous semantics. Moreover, by the definition above there are some SCA-ASM rule constructors for computation/coordination presented previously in Table 1 that are not yet supported. These rule constructors are: let-rule, iterate-rule, choose-rule, split-rule and spawn-rule. We postpone as future work the extension of the model to include also these rule constructors.

We denote by *root* the root of the *RDT* and by *PAR* the set of nodes labeled by `par`. For a node $v \in V$, we denote by $d(v)$ the set of child nodes of v and by $L(v)$ the set of leaf nodes of the tree rooted at v.

For the internal nodes of the RDT, we define the labeling function:

$$t : V \rightarrow \{seq, par, if, forall\}$$

For the edges of the RDT, we introduce the labeling function:

$$w : E \rightarrow [0, 1]$$

where $w(f(i), i)$, for each edge $(f(i), i) \in E$, yields the probability that the rule i is executed within rule $f(i)$. Such probability values are provided by the tuple $(P(rule_1), P(rule_2), \ldots, P(rule_m))$ of the usage profile for the SCA-ASM component (see Definition 6).

The RDT corresponding to the SCA-ASM model of the bill service component (see Fig. 4) reported in Listing 1, is shown in Fig. 5. Unlabeled edges in the figure have a probability value equal to 1. By visiting the RDT in preorder, we have associated the labels v, $1 \leq v \leq |V| = 9$, to the nodes.

Note that an RDT can be defined for any named rule of M, but we treat such trees as subtree of the "main RDT" by unfolding the nodes corresponding to their invocations (i.e. rule call nodes).

We denote by $v \prec v'$ if the node v is a *descendant* of the node v'. We also introduce a new relation between nodes, as follows.

Definition 9. Given an RDT (V, E) of an SCA-ASM component M, we say that a node $v \in V$ is a *direct PAR descendant* of $v' \in V$, if $v \prec v'$ and for any other node $v'' \in V$, $v \prec v'' \prec v'$ implies $v'' \neq par$, i.e., if there is no node `par` rule in the path from v to v'. We denote by $v \preceq_{dd} v'$ the node v as *direct PAR descendant* of v'.

3.3. Reliability model formulation

Since we assume that for each service k provided by the SCA assembly there exists an SCA-ASM core component M_k, the reliability of an SCA assembly related to the execution of the service k is computed by exploiting the rule dependency tree $RDT_k = (V_k, E_k)$ of M_k. To this purpose, we exploit the notions of *move* (a single computation step) and of *run* of an SCA-ASM component related to the invocation of the service, as defined in Section 2. These definitions reflect the *in-place simulation* semantics (Brugali et al., 2011) of SCA-ASM models within an SCA-compliant runtime platform.

For each service k provided by the SCA application, considering the (logarithm of the) reliability is additive (Cardoso et al., 2004), the reliability of the move mov_i of M_k is obtained by combining the reliability of the SCA components involved in the provision of k with the reliability of the SCA-ASM core component M_k itself that orchestrates such components. Formally:

$$Rel^i_{M_k} = e^{\left(\sum_{c=1}^{|C_k|} n^i_c \cdot r^i_c\right) + \bar{r}^i_{M_k}} \quad (1)$$

Under the simplifying assumption of independence between failures, $Rel^i_{M_k}$ depends on:

- the set $C_k \subset C$ of components orchestrated by the core SCA-ASM component for providing the service k;
- the probability n_c^i that the component $c \in C_k$ is invoked;
- the logarithm of the reliability r_c^i of the SCA component c, namely the probability that the components completes its task when invoked;
- and the logarithm of the reliability $\bar{r}_{M_k}^i$ of the core SCA-ASM component, which reflects the nature of failures in an ASM.

The parameter n_c^i can be easily estimated by parsing the paths of the RDT_k of the core component, from the root to the leafs containing primitives `wsendreceive`/`receive` to/from c, and multiplying the w_i labels of the arcs along the paths.

To compute the reliability of SCA-ASM components – depending on failures specific to an ASM –, and thus the value $\bar{r}_{M_k}^i$, we provide a precise method in Section 4. Precisely, the reliability $\bar{r}_{M_k}^i$ of an SCA-ASM component related to a single move can be estimated by the formula (4), while the reliability r_c of an SCA-ASM component over an entire run of moves is estimated by the formula (5).

In case the component c is not an SCA-ASM component, its reliability r_c must be given by the user. Suggestions to estimate r_c can be found in Cortellessa et al. (2006). A rough upper bound $1/N_{nf}$ can be estimated upon observing that the component has been invoked for N_{nf} number of times with no failures.

For the sake of model linearity, as in Zeng et al. (2004), we consider the *natural logarithm* of the reliability rather than the reliability itself. Note that we defined the parameters of components as average values of the values of their provided services. They are defined on the basis of the usage profile and could be refined with respect to the services without essentially changing the structure of the overall reliability model.

Now, for each provided service k, the reliability Rel_{M_k} of a run_k of length n of M_k, under failures independence assumption, can be modeled as follows:

$$Rel_{M_k} = \prod_{i=0}^{n-1} Rel_{M_k}^i \tag{2}$$

Finally, in an average case setting (Goseva-Popstojanova and Trivedi, 2001), considering all provided services $k \in K$ of the SCA assembly and the SCA-ASM core components M_k for such services, the reliability of the overall assembly SCA is:

$$Rel = \sum_{k=1}^{|K|} pexec_k \cdot Rel_{M_k} \tag{3}$$

4. Reliability of an SCA-ASM component

We show now how to compute the reliability of an SCA-ASM component by considering failures specific to the semantics of ASMs. As ASM crash failures we here consider the yielding of an *inconsistent update* set and the violation of *invariants*. In particular, note that the function updates that we consider in our approach are of the form $f(t_1, \ldots, t_n) := t$ where t is a function term whose value (interpretation) depends on a monitored function, and therefore on the environment and not only on the controlled part of the machine. In such scenario, our reliability prediction method is useful because it helps to reason about these non trivial inconsistent updates that cannot be determined by conventional functional validation and verification techniques.

To compute the reliability of an SCA-ASM component, we still exploit the notions of RDT, *move*, and *run* of an SCA-ASM component.

Under failures independence assumption, the reliability of a move mov_{i_c} of an SCA-ASM component c, is:

$$Rel_c^i = Rel_{IU}^i \cdot Rel_{I_M}^i \cdot Rel_{I_{env}}^i \tag{4}$$

where Rel_{IU}^i is the SCA-ASM reliability for *inconsistent update failures*, whereas $Rel_{I_M}^i$ and $Rel_{I_{env}}^i$ are the SCA-ASM machine and the environment reliabilities, respectively, for *invariant failures*.

The reliability r_c of an SCA-ASM component c over a run of n moves can be modeled as:

$$r_c = \prod_{i=0}^{n-1} Rel_c^i \tag{5}$$

4.1. Invariant failures (InvF)

We here give in a parametric way the definition and the exemplification for the environment (*env*) reliability $Rel_{I_{env}}^i$ and for the machine (M) reliability $Rel_{I_M}^i$. For convenience, we distinguish state invariants in: *controlled invariants*, invariants that involve only controlled/out functions, and *monitored invariants*, invariants that involve monitored functions (not only controlled/out) and those shared functions the component never updates (therefore they can be treated as monitored for reliability estimation).

Let Z_{env} be the set of monitored invariants and Z_M be the set of controlled invariants.

Example 1 *(Z_{env} and Z_M estimation).* Let us consider the SCA-ASM model of the `BillServiceComponent` (see Fig. 4 and Listing 1). Fig. 5 shows the RDT of the component's program *r_BillServiceComponent*. For this component, it yields $Z_{env} = \{tipRate\_inv\}$[8] and $Z_M = \{priceWithTaxRate\_inv, usrTipRate\_inv\}$.

4.1.1. Environment reliability in presence of invariant failures

Under the assumption that the failures are independent and that monitored invariants are evaluated at the beginning of the state transition mov_i, before firing the rules of M enabled in the current state s_{i-1}, the reliability $Rel_{I_{env}}^i$ can be formulated as follows:

$$Rel_{I_{env}}^i = \prod_{z \in Z_{env}} (1 - P(A_z^i)) \tag{6}$$

where A_z^i is the event "The invariant z is violated at the move i due to an update by the *env*".

The probability $P(A_z^i)$ can be computed in terms of the distribution of the $P((m^i)_j)$ at the move i of the component (see Definition 7). This distribution comes together with the usage profile. Formally, it yields

$$P(A_z^i) = \prod_{m \in Mon(z)} (1 - P(m^i)) \tag{7}$$

where $Mon(z)$ is the set of monitored functions[9] occurring in the invariant z and $P(m^i)$ the probability that the function m is erroneously updated by the environment.

To provide an operational support to the computation of $Rel_{I_{env}}^i$, we report the algorithm in Fig. 6.

Example 2 *($Rel_{I_{env}}^i$ estimation).* Let us assume that the monitored invariant *tipRate_inv* of the running example is the *only* model

[8] Although function *tipRate* is declared as shared in the SCA-ASM module, the `BillServiceComponent` considers it as monitored since it never updates the function value.

[9] This set includes also those shared functions the component never updates and that therefore they can be treated as monitored for reliability estimation.

Algorithm Compute$Rel^i_{I_{env}}$
1. $Rel^i_{I_{env}} \longleftarrow 1$
2. **for all** $z \in Z_{env}$ **do**
3. $\quad Rel^i_{I_{env}} \longleftarrow Rel^i_{I_{env}} \cdot (1 - P(A^i_z))$
4. **end for**

Fig. 6. Computation of $Rel^i_{I_{env}}$.

monitored invariant, and let us assume (see Eq. (6)) that $Rel^i_{I_{env}} = (1 - P(A^i_{tipRate\_inv}))$ yields 0.92. Upon introducing a new invariant, the reliability $Rel^i_{I_{env}}$ may decrease. Let us suppose to add the monitored invariant

invariant *monAB_inv* ***over*** *monA*, *monB* : $monA = \neg monB$

over two additional monitored functions *monA* and *monB* to the SCA-ASM component specification. It now yields $Z_{env} = \{tipRate\_inv,\ monAB\_inv\}$. If $(1 - P(A^i_{monAB\_inv}))$ would be equal to 0.97, then the reliability $Rel^i_{I_{env}}$ obtained as a functions of the $P(A^i_z)$ probabilities of the two invariants, decreases from 0.92 to 0.8924.

4.1.2. Machine reliability in presence of invariant failures

To compute the reliability $Rel^i_{I_M}$, for the sake of the failure independence requirement, we here assume that the expression of a controlled invariant may be influenced by at most one update rule of *M*. The association of the invariants to the update rules that influence them is represented by the matrix $RI_M(|L(root)| \times |Z_M|)$, where an element $RI_M[v, z]$ is equal to 1 if the dynamic functions occurring in the invariant *z* are modifiable by the rule *v*, 0 otherwise.

The reliability $Rel^i_{I_M}$, under the assumption that failures are independent and the values of the functions read by *M* (i.e., monitored functions) are correctly provided by the *env* (i.e., without violating monitored invariants), can be modeled as follows:

$$Rel^i_{I_M} = e^{\psi^i} \tag{8}$$

The parameter ψ^i can be estimated as follows:

$$\psi^i = \sum_{v \in L(root)} p_v \cdot \sum_{z \in Z_M} RI_M[v, z] \cdot ln(P(B^i_{zv}|I^i_v))$$

where

ln is the *natural* logarithm;
p_v is the probability that the rule *v* is invoked, obtained from the RDT multiplying the labels of the edges along the path from the root to rule *v*;
B^i_{zv} is the event "functions occurring in the invariant *z* are correctly updated by *M* executing *v* in the mov_i";
I^i_v is the event "the values of monitored functions occurring in the rule *v* are correctly updated by the environment at mov_i".

The probability $P(B^i_{zv}|I^i_v)$ can be computed in a way similar to the computation of $P(A^i_z)$ in (7) by using the information of the usage profile (see Definitions 6 and 7)

Fig. 7 reports the algorithm to compute the reliabilities $Rel^i_{I_M}$.

Example 3 *($Rel^i_{I_M}$ estimation).* The invariants *priceWithTaxRate_inv* and *usrTipRate_inv* of the running example must be verified for rules $v = 4$ and $v = 2$, respectively. As an example, let us consider that the probability $P(B^i_{zv}|I^i_v)$ of the rules 4 and 2, at a certain move *i*, is set to 0.94 and 0.97, respectively. Thus the reliability $Rel^i_{I_M}$ using formula (8) is equal to 0.911800. The reliability $Rel^i_{I_M}$ may increase depending on the variation of the probability that an invariant is satisfied for a rule. For example, if we increase $P(B^i_{z4}|I^i_4)$ from 0.94 to 0.989,

Algorithm Compute$Rel^i_{I_M}$
1. $\psi^i \longleftarrow 0$
2. **for all** $(v = 1, \ldots, |L(root)|)$ **do**
3. $\quad$ **for all** $(z = 1, \ldots, |Z_M|)$ **do**
4. $\quad\quad \psi^i \longleftarrow \psi^i + (p_v \cdot RI_M[v, z] \cdot ln(P(B^i_{zv}|I^i_v))$
5. $\quad$ **end for**
6. **end for**
7. $Rel^i_{I_M} \longleftarrow e^{\psi^i}$

Fig. 7. Computation of $Rel^i_{I_M}$.

then $Rel^i_{I_M}$ increases to 0.959330. Therefore the probability $P(B^i_{zv}|I^i_v)$ (combined with the probability p_v) of the rules may sensibly affect the reliability.

4.2. Inconsistent update failures (IU)

The reliability Rel^i_{IU} of an SCA-ASM component is defined as the probability that no inconsistent update is performed for a specific move mov_i.

Let *I* be the event "the input of the SCA-ASM is correct, i.e., the monitored/shared functions updated by the *env* do not violate state invariants", and *O* the event "no inconsistent update is ever performed". Then the reliability Rel^i_{IU} can be expressed as follows:

$$Rel^i_{IU} = P(I \cap O) = P(I)P(O|I) \tag{9}$$

where $P(I) = 1$ since we assume *I* is a certain event.

The parallel execution of rules may generate an inconsistent update, thus Rel^i_{IU} depends on the probabilities $P(O_v|I)$ with $v \in PAR$ that the `par` rules do not perform inconsistent updates, where O_v is the event "no inconsistent update is ever performed by the `par` rule $v \in PAR$".

Example 4 *(IU Running Example).* Fig. 8 shows an RDT to exemplify the *IU* failures. By visiting it in preorder, we have associated with the nodes labels assuming values $v \in [1, 28]$. Symbols *e*, *q*, *t*, and *u* denote monitored functions.

In the remainder of this section we first show how to compute the probability $P(O_v|I)$ and then how to compute $P(O|I)$ to model the whole reliability Rel^i_{IU}.

4.2.1. Probability "$P(O_v|I)$" estimation

For a parallel rule $v \in PAR$, the probability $P(O_v|I)$ can be obtained recursively by a depth-first traversal of the *RDT*. Let C_v be the event "an inconsistent update is performed among the subtrees of the child nodes of *v*", and $O_{v'}$ be the event "no inconsistent update is ever performed by $v' \preceq_{dd} v''$". Then $P(O_v|I)$ is[10]:

$$P(O_v|I) = (1 - \mu_v) \cdot e^{\phi_v} + \mu_v \cdot (1 - P(C_v|I)) \tag{10}$$

where the parameter ϕ_v can be expressed as follows:

$$\phi_v = \sum_{v' \in PAR,\, v' \preceq_{dd} v} p_{v'} \cdot ln(P(O_{v'}|I)) \tag{11}$$

$P(O_v|I)$ is the sum of two terms. The first term depends recursively on the probabilities $P(O_{v'}|I)$ that the *direct PAR descendant* nodes v' of *v* – corresponding to outer `par` rules not nested within other `par` rules – do not perform an inconsistent update. Thus, this term is obtained by visiting the *RDT* in postorder and properly aggregating the probability to perform an inconsistent update

[10] Note that the number of child nodes of a `forall` rule is not fixed and depends on the operational profile.

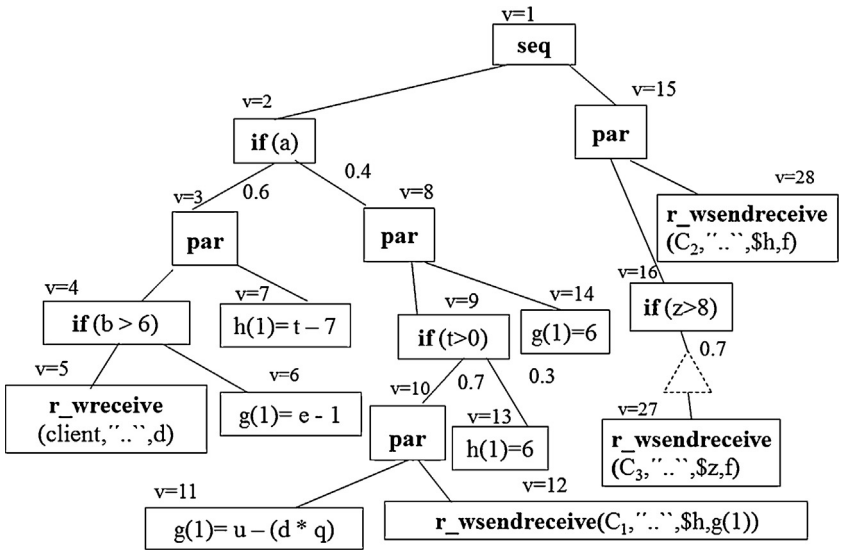

Fig. 8. RDT for the *IU* example.

of the child nodes. The second term is a function of the probability $P(C_v|I)$ that the subtrees of the child nodes of v perform an inconsistent update among each other.

Example 5 *($P(O_v|I)$ estimation).* The probability that no inconsistent update is ever performed by the `par` rule $v = 8$ depends on: (1) the probability that its *direct PAR descendant* node $v = 10$ does not perform an inconsistent update of the function $g(1)$; and the probability that the subtrees rooted at the child nodes $v = 9$ and $v = 14$ perform an inconsistent update of $g(1)$ among each other.

Let us now detail the parameters of the right-side of (10) and (11) as follows:

μ_v expresses the conditional probability that a single execution of the rule v does not fail (i.e., the whole SCA-ASM does not crash before the end of the execution of the rule v) on a test case following a certain input distribution. It can be derived by using the guidelines of the *testability* (Voas and Miller, 1995) estimations. In fact, this property expresses the conditional probability that a single execution of a software fails on a test case following a certain input distribution. In Cortellessa et al. (2006), we suggest a procedure to estimate it.

Example 6 *(μ_v estimation).* For example, μ_3 is the probability that the `par` rule $v = 3$ is executed without stopping for a failure raised by either the child nodes 4 or 7.

$p_{v'}$ is the probability that the node v' is invoked from the node v, obtained by multiplying the labels of the arcs along the path from the node v to v'.
$P(C_v|I)$ is the probability that the subtrees of the child nodes of v perform an inconsistent update among each other.

Example 7 *($P(C_v|I)$ estimation).* $P(C_{10}|I)$ is the probability that the nodes 11 and 12 generate an inconsistent update for the function $g(1)$, whereas $P(C_{15}|I)$ is the probability that the subtrees of the child nodes 16 and 28 generate an inconsistent update among each other for the function f. Therefore, $P(C_{10}|I)$ is the probability that the nodes – the SCA-ASM component itself and the component C_1 – write differently $g(1)$. $P(C_{15}|I)$ is the probability that components C_3 and C_2 update differently f.

Formally:

$$P(C_v|I) = 1 - e^{\alpha_v} \tag{12}$$

The parameter α_v is:

$$\alpha_v = \sum_{l_1 \equiv (l,x) \in L(v'), l_2 \equiv (l,y) \in L(v''), v', v'' \in d(v)} p_{l_1} \cdot p_{l_2} \cdot \beta_{l_1 l_2} \tag{13}$$

and, the parameter $\beta_{l_1 l_2}$ is:

$$\beta_{l_1 l_2} = ln(1 - P(x \neq y)) \tag{14}$$

where (l, x) and (l, y) are the updates produced by the leaf nodes $l_1 \in L(v')$ and $l_2 \in L(v'')$, respectively, for the same location (dynamic function) l of the SCA-ASM signature, and $d(v)$ is the set of the child nodes of v. Thus, upon estimating the formula (14) for each pair of update rules l_1 and l_2, we substitute this estimation in the formula (13). Finally, by back substituting in formulas (12), we obtain an expression for $P(C_v|I)$.

$\beta_{l_1 l_2}$ can be estimated as follows. A rough upper bound $1/Nnf$ of $\beta_{l_1 l_2}$ can be obtained by monitoring (Gargantini and Riccobene, 2001) the rules l_1 and l_2 and observing their execution for a Nnf number of times with no failures (i.e., an inconsistent update is not performed). Alternatively, $\beta_{l_1 l_2}$ can also be estimated with the formulas introduced in Abdelmoez et al. (2004) for the *error propagation* probability from component A to component B, and the ones in Abdelmoez et al. (2005) for the *change propagation* probability. To take into account also the probability of failure on demand of a component in the error propagation the approach in Filieri et al. (2010) can be applied.

The parameters μ_v, $p_{v'}$, and $P(C_v|I)$ may be characterized by a not negligible uncertainty. The propagation of this uncertainty should be analyzed, but it is a very complex task and it is outside the scope of this paper. Several methods to perform this type of analysis can be found in the literature, as it has been done, for example, in Goseva-Popstojanova and Kamavaram (2004) for a reliability model.

Example 8 *($P(O_v|I)$ variations).* Let us consider the following values for the `par` node v=8: the probability μ_v and $P(C_v|I)$ equal to 0.6 and 0.08, respectively, and the probability $P(O_{10}|I)$ = 0.97. Thus by applying formula (10) the probability $P(O_8|I)$ is equal to 0.943561.

The probability $P(O_v|I)$ may decrease depending on the variation of the probability that the single execution of the *direct PAR*

descendant nodes does not generate an inconsistent update. As an example, while changing the component C_1's features, if $P(O_{10}|I)$ decreases from 0.97 to 0.84, then $P(O_8|I)$ decreases from 0.943561 to 0.906042.

4.2.2. Probability "P(O|I)" estimation

The overall expected probability $P(O|I)$ depends on the type of the *root* of the RDT and can be easily defined by the following expression:

$$P(O|I) = \begin{cases} 1 & t(root) \in \{skip, update\} \quad \text{(i)} \\ P(O_{root}|I) & t(root) \in \{par, forall\} \quad \text{(ii)} \\ e^{\phi_{root}} & t(root) \in \{if, seq\} \quad \text{(iii)} \end{cases} \tag{15}$$

If the *root* is a par rule (case (ii) in the formula (15)), then $P(O|I)$ is equal to $P(O_{root}|I)$ (see equation (10) in Sect. 4.2.1). In the last case (iii), $P(O|I)$ is equal to ϕ_{root} (see formula (11) in Sect. 4.2.1) because the updates are disjointed and thus $P(O|I)$ depends only on the probability $P(O_v|I)$ that the root's *direct PAR descendant* nodes do not perform inconsistent updates.

Example 9 *(Rel^i_{IU} estimation).* Starting from the value $P(O_8|I)$ = 0.943561 obtained in the previous example, if for the `par` nodes 3 and 15 the probabilities $P(O_v|I)$ are equal to 0.87 and 0.62, respectively, then the reliability Rel^i_{IU} is equal to 0.557200.

The reliability Rel^i_{IU} may increase depending on the variation of the probability that the single execution of the *direct PAR descendant* nodes does not generate an inconsistent update. For example, if $P(O_{15}|I)$ increase to 0.87, then Rel^i_{IU} increases to 0.781877. Finally, if $P(O_3|I)$ and $P(O_{15}|I)$ increase to 0.94 and 0.98, respectively, then Rel^i_{IU} increases to 0.922594. This highlights how our model may help to combine (and, in general, to reason about) the decisions to be taken for each component. As we have shown in this case, the architecture decisions might have to be modified while changing the software features.

In order to provide an operational support to the model, we have plugged the previous formulas into an algorithm that estimates the reliabilities Rel^i_{IU}. The algorithm is illustrated in Figs. 9 and 10. A further algorithm for computing $P(C_v|I)$ is straightforward. It is similar to a recursive procedure that selects the leaf nodes of the

Algorithm ComputeRel^i_{IU}

1. **if** $t(root) \in \{$`skip`, `update`$\}$ **then return** 1
2. **if** $t(root) \in \{$`par`, `forall`$\}$ **then return** Compute$P(O_v|I)$(root)
   //$t(root) \in \{$`if`, `seq`$\}$
3. $\phi_{root} \longleftarrow 0$
4. **for all** $(v \in PAR, v \preceq_{dd} root)$ **do**
5. $\quad \phi_{root} \longleftarrow \phi_{root} + (p_v \cdot \text{Compute}P(O_v|I)(v))$
5. **end for**
6. **return** $e^{\phi_{root}}$

Fig. 9. Algorithm 1: computation of Rel^i_{IU}.

1. **function** Compute$P(O_v|I)$(node v)
2. $\phi_v \longleftarrow 0$
3. **for all** $(v' \in PAR, v' \preceq_{dd} v)$ **do**
4. $\quad \phi_v \longleftarrow \phi_v + (p_{v'} \cdot \text{Compute}P(O_v|I)(v'))$
5. **end for**
6. $P(O_v|I) \longleftarrow \mu_v \cdot (1 - P(C_v|I)) + (1 - \mu_v) \cdot e^{\phi_v}$
7. **return** $\ln(P(O_v|I))$

Fig. 10. Algorithm 2: computation of $P(O_v|I)$.

subtrees of the child nodes of v and searches in the collected update sets for updates competing to write the same location.

Note that the computation of $P(O_v|I)$ given by Algorithm 10 can be optimized by exploiting a tree traversal algorithm and using the notion of least common ancestor for the leaf nodes competing for the update of the same location.

5. The Finance case study

In this section, we show the results of applying the SCA reliability model to an application inspired by the "Finance case study" of the EU project SENSORIA. Readers interested in the application details, that we do not provide here, can refer to Banti et al. (2008). Shortly, a credit (web) portal application of a credit institute allows customer companies to loan from a bank. Fig. 11 shows the SCA assembly of the finance application. It consists of the following SCA components: `Portal` ($c = 1$), `Authentication` ($c = 2$), `InformationUpload` ($c = 3$), `Validation` ($c = 4$), `InformationUpdate` ($c = 5$), `RequestProcessing` ($c = 6$), and `ContractProcessing` ($c = 7$). Actors supervisor, employee and the

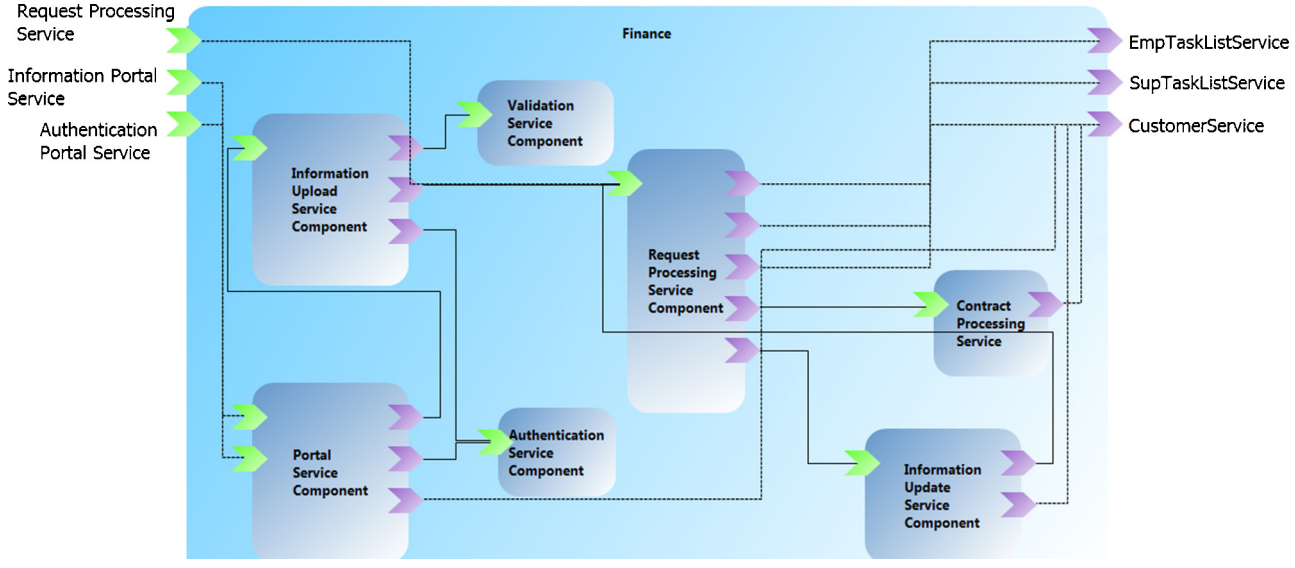

Fig. 11. SCA assembly of the Finance case study.

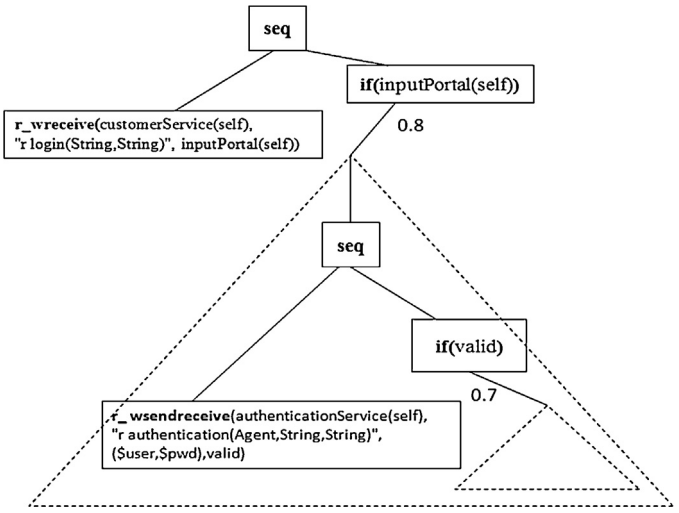

Fig. 12. RDT for the `AuthenticationPortalService` service.

customer itself (that starts the overall services scenarios) appear as external partners (see the promoted services and references of the SCA composite `Finance` in Fig. 11).

We have taken into account two services provided by the core SCA-ASM component `Portal`. The first service, `AuthenticationPortalService`, allows users to register to the portal; the second one, `InformationPortalService`, allows users to request a loan. Fragments of the RDTs corresponding to the SCA-ASM models of the `AuthenticationPortalService` and `InformationPortalService` are shown in Figs. 12 and 13, respectively.

After estimating the reliability of both services by the formula (2), the reliability of the whole application has been obtained by the formula (3), under the hypothesis that probabilities of the two service invocations yield 0.4 and 0.6, respectively.

The `Portal` component coordinates the components: `Authentication` to allow the user registration, authentication and login, `InformationUpload` to request a loan. For processing the credit request, this latter component interacts with the components: $c = 2$ for user authentication, $c = 4$ to involve a preliminary evaluation by an employee, $c = 6$ to allow a supervisor to examine and draft a contract by using the components $c = 5$ and 7. The `Authentication` and `InformationUpload` components are SCA-ASM components. Moreover, we assume that for these components the reliability depends only on ASM failures (i.e., the invariant failures *InvF* and the inconsistent updates failures *IU*). In order to obtain a trustworthy estimation of the reliability model for the Finance case study the Rel^i_{IU}, $Rel^i_{I_M}$ and $Rel^i_{I_{env}}$ in the formula (4) have to be considered.

We have conducted some experiments on the Finance case study that differ mainly in the SCA-ASM reliability parameters – for the probability to perform inconsistent updates and violate invariants – of the single SCA components and of the core components. We here focus on the exposed service `AuthenticationPortalService` and on the core component `Portal` for the considered service. We report below the results of three incremental experiments corresponding, respectively, to the following three configurations:

First configuration We assume the reliability of the component `Authentication` depends only on invariants failures and its value varies from 0.89 to 1. We also assume the reliability of the component `InformationUpload` depends on *IU* failures and it varies in the range [0,0.3].
Second configuration In addition to the first configuration scenario, we consider also the reliability value of `InformationUpload` for invariants failures fixed to 0.91.
Third configuration With respect to the previous two configurations, the reliability of the core component `Portal` is also considered by fixing its value to 0.92.

More details on the configuration parameters and on the formulas used in these experiments are reported in the paragraphs below together with the experimental results.

First experiment: considering the InvF and IU failures separately for elementary SCA-ASM components. We have observed the reliability of the application while varying the reliability of `Authentication` – assumed to be due only to *InvF* (i.e., $Rel^i_{I_{env}}$ and $Rel^i_{I_M}$) – and varying the reliability of `InformationUpload`. We have assumed that this last reliability is due, other than to *InvF* of `Authentication`, to *IU*

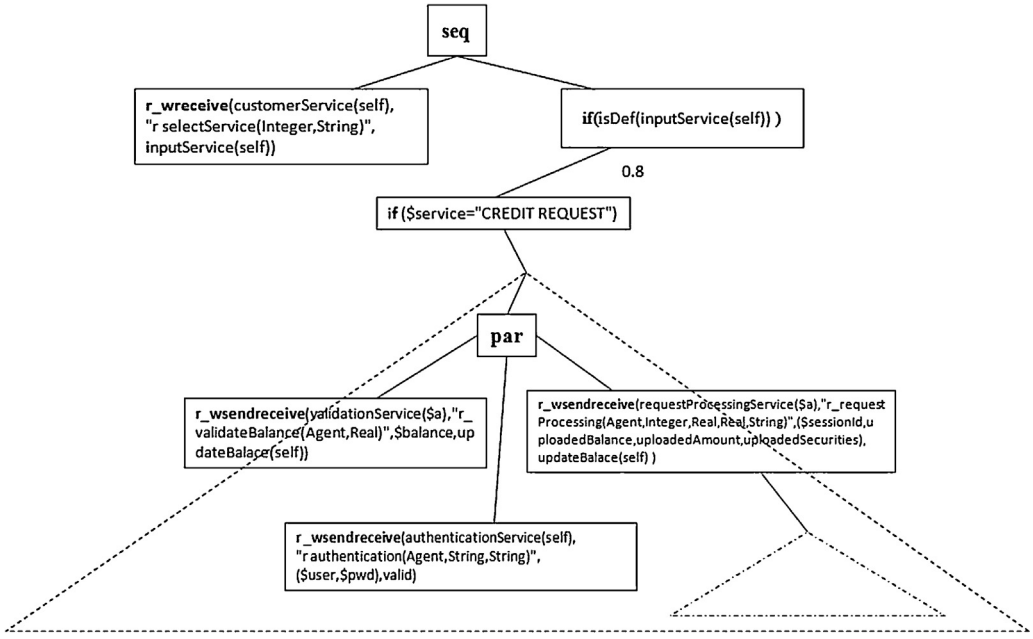

Fig. 13. RDT for the `InformationPortalService` service.

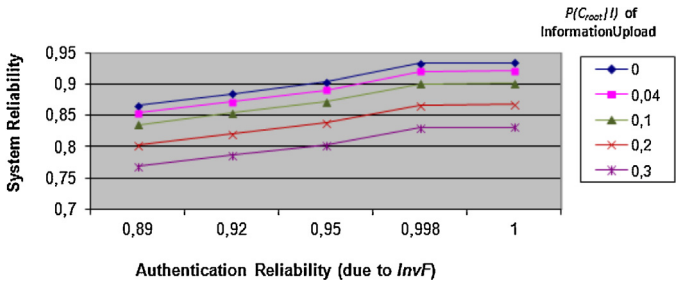

Fig. 14. Reliability analysis results for the first configuration.

failures (i.e., Rel^i_{IU}), which may be performed by the components $c=4$ and $c=6$, and thus, consequently, by the components used by these components (i.e., c equal to 5 and 7). So, we assume that the *InvF* and *IU* failures are considered only for the application's elementary SCA-ASM components and not for the core component itself.

In Fig. 14 we report the results. Each curve represents the reliability of the application while varying from 0.89 to 1 the reliability of `Authentication` – due to *InvF*. Curves differ because a different fixed value for the parameter $P(C_{root}|I)$ (where C_{root} is the event "an inconsistent update is performed among the subtrees of the child nodes of the root node", see Section 4.2.1) of `InformationUpload` has been assigned for each curve. We have obtained the curves by varying this last value from 0 to 0.3.

For a given value of the probability $P(C_{root}|I)$ of `InformationUpload` (i.e., for a given curve), the reliability of the application increases while increasing the reliability of `Authentication`. This provides a first evidence of the relevance of the *InvF* and *IU* failures in our SCA-ASM reliability model. In fact, the assumption of not considering the *InvF* for `Authentication`, corresponds in Fig. 14 to the points of the curves where the reliability of `Authentication` on the x-axis equals 1. As opposite, the assumption of not considering the *IU* failures for `InformationUpload`, corresponds in Fig. 14 to the curve where $P(C_{root}|I)$ of `InformationUpload` is equal to 0. Such assumptions correspond, for each curve, to the maximum values of the application reliability. This means that the no consideration of failures brings to an overoptimistic prediction of the application reliability. This result confirms that the *InvF* and *IU* failures analysis are a key factor for a trustworthy reliability prediction, and its estimation leads in our model a more precise (and less optimistic) estimation of the reliability.

On the other hand, for the same value for the reliability of `Authentication`, the application reliability decreases while increasing the value of $P(C_{root}|I)$ for `InformationUpload`. This can be observed by fixing a value on the x-axis and observing the values on the curves while growing $P(C_{root}|I)$.

Second experiment: considering both InvF and IU failures for an elementary SCA-ASM component. As in the first configuration, we observed the application reliability while varying the reliability of `Authentication`, assumed due only to *InvF*, and varied this last value from 0.89 and 1. Moreover, other than varying from 0 to 0.3 the value of $P(C_{root}|I)$ for `InformationUpload`, we also considered the *InvF* for `InformationUpload`. We obtained the curves by setting this last value (i.e., $Rel^i_{I_{env}} \cdot Rel^i_{I_M}$) to 0.91.

Fig. 15(i) reports the results. Each curve represents the reliability of the application while varying the reliability of `Authentication` due to *InvF*, and the value for $P(C_{root}|I)$ of `InformationUpload`.

The introduction of the *InvF* for `InformationUpload` itself allows obtaining a better estimation of the application reliability. In fact, for corresponding values of the curves in the first and second experiments, the application reliability is lower in the second experiment than in the first one. This is because in the first experiment we consider the *InvF* only for the `Authentication` component, whereas in the second experiment we consider such a kind of failures also for `InformationUpload` itself.

Third experiment: considering also the reliability of the core component. Starting from the same initial configuration of the second experiment, we have also considered the failures of the core SCA-ASM component `Portal` for the service `InformationPortalService` provided by the application. In particular, we have considered this last value (i.e., $\bar{r}^i_{M_k}$ in formula (1)) equal to 0.92. Fig. 15(ii) reports the results. Each curve represents the reliability of the application while varying the reliability of `Authentication`, and the value for $P(C_{root}|I)$ of `InformationUpload`. A relevant observation is that, for corresponding values of curves in the second and third experiments, the reliability is higher in the second experiment than in the third one. This is because in the second experiment we consider the failures only for elementary SCA-ASM components, whereas in the third experiment we consider such failures also for the core SCA-ASM component of the application service. This highlights the novelty and capabilities of our approach. In fact, this difference would have not been perceived by using the existing approaches that typically predict the application reliability by only considering the reliability of single elementary components. As opposite, we predict the application reliability by composing

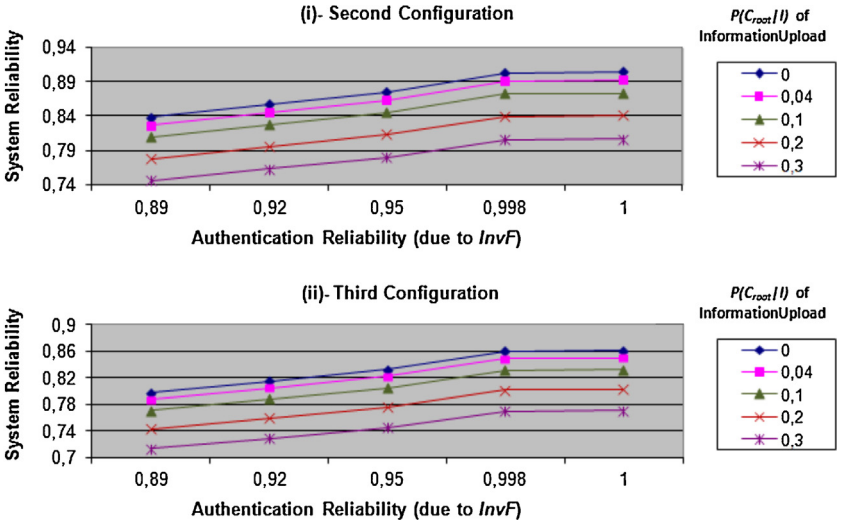

Fig. 15. Reliability analysis results for the second and the third configurations.

such kind of application failures with the ones due to the nature and stateful behavior of the core component itself.

6. Reliability model evaluation

In this section we compare the results that can be obtained with our reliability model and a state-of-the art model for the web service composition based on the use of BPEL (Alves et al., 2006). We first briefly introduce the BPEL-based approach by outlining the differences and similarities with the proposed SCA-ASM method and then we illustrate some experimental results.

6.1. BPEL-based reliability model

From the service oriented perspective, a business process is a means to have services interact to satisfy specific requests. This can be achieved through BPEL (Alves et al., 2006) orchestration. In particular, let S be a service-based system composed by n elementary web services (called *abstract services*), with s_i the ith abstract service ($1 \leq i \leq n$). Through the composition of its elementary software services, the BPEL composition offers K *external services* to users. With $pexec_k$ we denote the probability that the k-th system external service will be invoked. Following Cardellini et al. (2007), we restrict the attention to a significant subset of the whole BPEL definition, focusing on its structured style of modeling. Specifically, besides the primitive "invoke activity", which specifies the synchronous or asynchronous invocation of a Web service, we consider the *structured activities*: *Sequence*, *Switch*, *While* and *Flow*.

A BPEL tree can be seen as a directed acyclic graph $DAG_k = (V_k, E_k)$ representing the kth BPEL external service ($1 \leq k \leq K$). The nodes V_k are BPEL activities, while the edges E_k reflect the relationships among the BPEL activities (Cardellini et al., 2007).

For each abstract service, several *concrete services* may exist that match their description but that may differ for cost and reliability characteristics. We call J_i the set of instances for s_i, while s_{ij} represents the jth instance of s_i in J_i. A *service broker* acts as an intermediary for the matching between the abstract services and the concrete services - between service requestors and providers discovering and selecting the best concrete services in order to minimize the costs and guarantee a given level of system reliability (Cardellini et al., 2007). Hereafter, for simplicity and brevity, we only consider the service selection task. Fig. 16 shows the reliability model for the web service selection optimization problem. In particular, the figure shows how to predict the reliability of an external service, starting from its BPEL activity tree. Specifically, the reliability of the concrete service s_{ij} is combined with the expected number of times that the service s_i is invoked in the BPEL tree k. This latter parameter can be easily estimated by parsing the BPEL tree's paths – having the service i as leaf node – and multiplying the p_a labels by the arcs along the path. As far as the system reliability formulation is concerned, we have exploited the works in Cardellini et al. (2007) and Zeng et al. (2004). For the sake of model linearity, as in Zeng et al. (2004), when writing expressions, we will consider the logarithm of the reliability rather than the reliability itself.

SCA-ASM vs BPEL-based

Table 5 summerizes the symbols of the SCA-ASM reliability model. In Table 2, we compare the features of the RDT of the kth service provided by the SCA assembly and the BPEL activity tree of the kth service provided by the composite web service; while Table 3 shows the features of the reliability model of the kth service provided by the SCA assembly and the reliability model of the kth service provided by the BPEL model.

Table 2
Comparing SCA-ASM rule dependency trees and BPEL activity trees.

SCA-ASM rule tree $RDT_k = (V_k, E_k)$	**BPEL activity tree** $DAG_k = (V_k, E_k)$
V_k rules in the SCA-ASM model	V_k activities in the BPEL code
internal node rule constructor	*internal node* structured activity
leaf node a basic rule for computation (`skip` or update rule) or a communication rule	*leaf node* invoke activity (corresponds to an elementary service invocation)
w_i prob. of executing rule i in a rule constructor	p_a prob. of executing activity a in a structured activity

Table 3
Comparing the SCA assembly reliability model and BPEL-based reliability model

SCA assembly rel. model	BPEL-based rel. model
K number of services provided by the assembly	K number of services provided by the service composition
$pexec_k$ probability that the service k will be invoked	$pexec_k$ probability that the service k will be invoked
n_i^c prob. that the component c is invoked	inv_{ki} number of time that the service i is invoked
Rel_{M_k} reliability of the provided service k	Rel_k reliability of the provided service k

Table 4
Parameters of the available instances for the existing services.

Service ID	Service altern.	Reliability r_{ij}	Num. of inv. first scen. inv_{1i}	Num. of inv. second scen. inv_{2i}	Num. of inv. third scen. inv_{3i}
s_1	s_{11}	0.9993	3	2	2
	s_{12}	0.9994			
	s_{13}	0.99999			
s_2	s_{21}	0.996	3	2.084	3
	s_{22}	0.9995			
s_3	s_{31}	0.99985			1.2
s_4	s_{41}	0.999		1	
	s_{42}	0.99985			
	s_{43}	0.99999			
s_5	s_{51}	0.9993	1	1	1
	s_{52}	0.9998			
	s_{53}	0.99998			
s_6	s_{61}	0.94		1.084	
	s_{62}	0.9997			
	s_{63}	0.9999			
s_7	s_{71}	0.99	1	1.084	1
	s_{72}	0.992			
	s_{73}	0.999999			

1. Model Parameters and Variables

n	number of abstract services
J_i	concrete services available for the abstract service i
inv_{ki}	expected number of times that service i is invoked
q_{ij}	reliability on demand of j-th concrete instance
K	number of external service
$pexec_k$	probability that the k-th external service will be invoked
R	minimum reliability threshold
x_{ij}	concrete service j selected for the abstract service i

2. The (logarithm) of the reliability of the k-th external service:

$$\ln Rel_k = \sum_{i=1}^{n} inv_{ki} \sum_{j=1}^{J_i} x_{ij} \ln q_{ij}$$

3. In an average case scenario, the system reliability constraint:

$$\sum_{k=1}^{K} pexec_k Rel_k \geq R$$

Fig. 16. Reliability model using a BPEL-based approach.

6.2. Experimentation

Hereafter we show the numerical results obtained by comparing the SCA-ASM reliability model and the BPEL-based model referring to the illustrated case study. We first describe the generation of the model parameters and then we describe different experimental results.

6.2.1. Model parameters

Following a standard approach for parametric evaluation, we first define a set of so-called "nominal parameters" which constitute the starting point of the experimentation and then we generate random instances by perturbing the nominal values. The average, maximum and minimum values of the different obtained results are then considered. Table 4 shows the starting parameter values of the available instances for the abstract services. The second column of Table 4 lists the set of alternatives for each existing service. For each alternative: the reliability r_{ij} is given in the third column; the expected number of times inv_{1i} that the service i is invoked within the first external service is given in the fourth column; the expected number of times inv_{2i} that the service i is invoked within the second external service is given in the fifth column; the number of times inv_{3i} that the service i is invoked within the third external service is given in the sixth column.

Starting from the services' nominal values, we have generated 186 different system configurations (here also called *perturbed configurations*) by randomly changing the parameters. Two of these configurations differ for concrete services' reliabilities, which we have slightly decreased/increased (e.g., within 10% of the nominal values).

In order to generate the perturbed configurations, we have generated five concrete services bases. For each abstract service, the number of concrete services spans from 13 to 65 (i.e., related to one of the five services bases). Two service bases differ for the number – spanning from 13 to 65 – and the parameters of the perturbed configurations (i.e., one perturbed configuration corresponds to seven concrete services generated for the seven abstract services). For sake of result robustness, the service bases' parameters have been varied.

6.2.2. Numerical results

Case 1: In Fig. 17 we report the results obtained applying the SCA assembly model without orchestrators and the BPEL-based reliability model to the perturbed configurations. We have fixed the probabilities (i.e., $pexec_k$) that the first, second and third external

Table 5
Table of symbols used in the reliability model.

Symbol ID	Symbol specification
$RDT = (V, E)$	Rule Dependency Tree of an SCA-ASM M. V is the set of nodes labeled by the rules in the main rule, and E are the edges that reflect the nesting relationship among these rules.
$root$	Root of the RDT.
PAR	Set of nodes corresponding to `par` rules.
Z_{env}	Set of monitored invariants where functions updated by env occur.
Z_M	Set of controlled invariants where functions updated by M occur.
A^i_z	The event "The invariant z is violated at mov_i due to an update by the env".
RI_M	$\|V\| \times \|Z_M\|$ matrix, where an element $RI_M[v, z]$ is equal to 1 if the dynamic functions occurring in the invariant z are modifiable by the rule v, 0 otherwise.
w_i	Probability that the rule i is invoked.
$p_{v'}$	Probability that the rule v' is invoked from a rule v.
B^i_{zv}	The event "Functions occurring in the invariant z are correctly updated by M executing v in the mov_i".
I^i_v	The event "The values of monitored functions occurring in the rule v are correctly updated by the environment at mov_i".
I	Event "the input of the SCA-ASM is correct".
O	Event "no inconsistent update is ever performed".
O_v	Event "no inconsistent update is ever performed by the `par` rule $v \in PAR$.
C_v	Event "an inconsistent update is performed among the subtrees of the child nodes of v".
μ_v	Conditional probability that a single execution of the rule v does not fail.

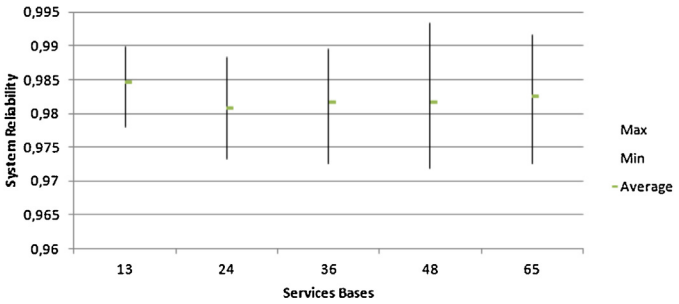

Fig. 17. System reliability obtained with the BPEL reliability model and with the SCA assembly without considering the orchestrators reliabilities.

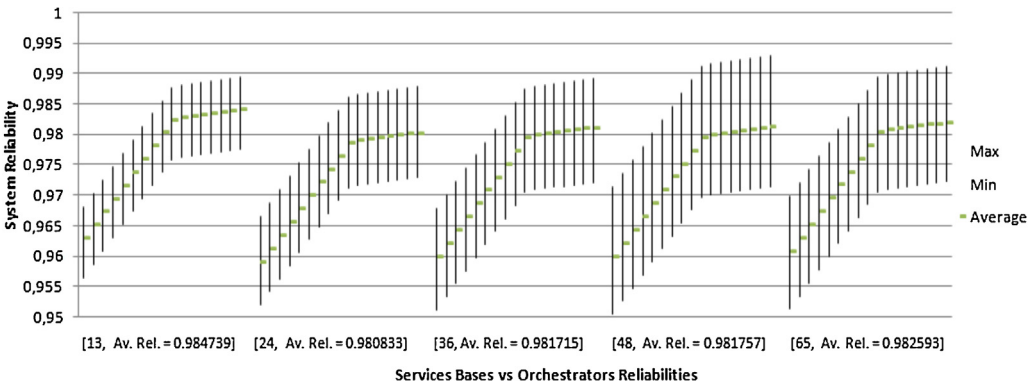

Fig. 18. System reliability obtained for the SCA assembly considering the orchestrators reliabilities.

service will be invoked with probabilities 0.3, 0.3, 0.4, respectively. Each bar – corresponding to one service base – contains the higher, lower and the average value obtained for the system reliability. The results are the same for both models.

Case 2: In Fig. 18 we illustrate the results obtained considering the SCA assembly taking into account also the service orchestrator's reliability.

For each perturbed configuration, we have applied our SCA assembly reliability model for a set of reliability values for the three external services' orchestrators. Each orchestrator reliability spans from 0.95 to 0.999 by steps of 0.005. Each group of seventeen bars – corresponding to one service base – refers to the execution model results over the base's perturbed configurations. In particular, each bar – corresponding to one fixed reliability value for the reliability of the three services orchestrators – contains the higher, lower and the average system reliability. The lower (higher or average) system reliability increases while increasing the orchestrators reliabilities. For example, with a service base of 13 perturbed configurations, if the reliability of the three orchestrators is equal to 0.95, the average system reliability decreases from 0.984739 (average value estimated without considering the orchestrators reliabilities in case 1) to 0.963045, whereas if the reliability of all orchestrators increases to 0.995, then the system reliability is about 0.982597.

These results show the relevance of the orchestrator' reliability in a reliability model.

Case 3: In order to compare the efficacy of considering the orchestrators reliabilities, we have analyzed in Fig. 19 the gain in reliability - in terms of a more precise (and less optimistic) estimation - obtained in three different experiments. For a perturbed configuration s, we have estimated the profit as follows: $(RelSys_{v}(s) - RelSys_{v'}(s))/RelSys_{v}(s)$, where $RelSys_{v}(s)$ and $RelSys_{v'}(s)$ represent the system reliability returned by the SCA assembly reliability model by considering the orchestrators reliabilities fixed to v and v', respectively.

The first experiment – corresponding to the curve with diamonds – refers to the execution of SCA assembly by considering the reliability of all orchestrators (i.e., the results obtained in case 2). The second experiment – corresponding to the curve with rectangles – refers to the execution of SCA assembly reliability model by considering the reliability of the first service's orchestrator equal to 1. Finally, the third experiment – corresponding to the curve with triangles – refers to the execution of SCA assembly reliability model where the reliability of the third service's orchestrator is equal to 1. The figure shows the results for the service base with 48 perturbed configurations. The x-axis represents the variation of the orchestrator reliability. Specifically, each point – corresponding to one reliability orchestrator's value – contains the average relative profit obtained for the system reliability.

A relevant observation is that for corresponding values of curves in the three experiments, the average profit is higher in the first experiment than in the second and third ones. This is because in the first experiment we consider the reliabilities of all services orchestrators, whereas in the second and third experiment we fix to 1 the reliability of one of the service orchestrators. For example, for the first experiment, with the reliabilities of the orchestrators equals to 0.97, the (modulus) average profit is about 0.01332, whereas for the second and third experiment, the (modulus) average profit is about 0.00928 and 0.00795, respectively.

Case 4: Fig. 20 shows the results we obtain when in the SCA assembly we explicitly model the reliability of an SCA-ASM component considering the service base of 65 perturbed configurations.

Fig. 20(a) illustrates the results obtained by explicitly using for s_3 our SCA-ASM reliability model. We have observed the system reliability while varying the reliability of s_3 – assumed to be due to

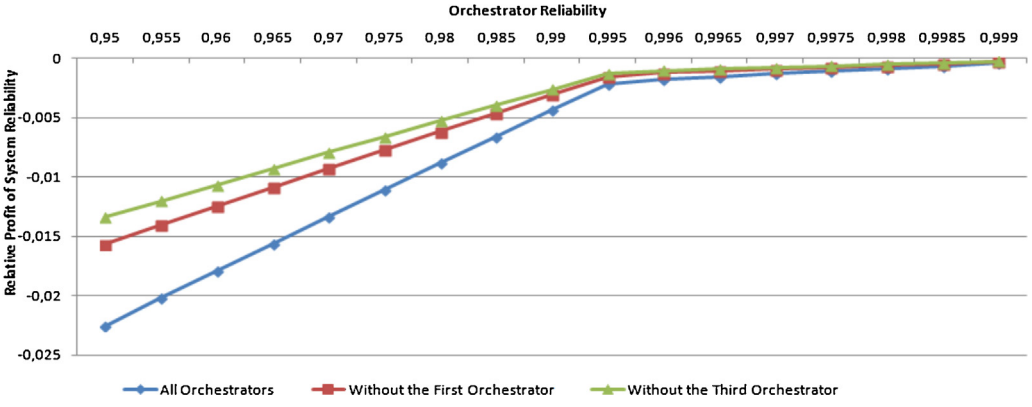

Fig. 19. Comparing the relative profit of the system reliability.

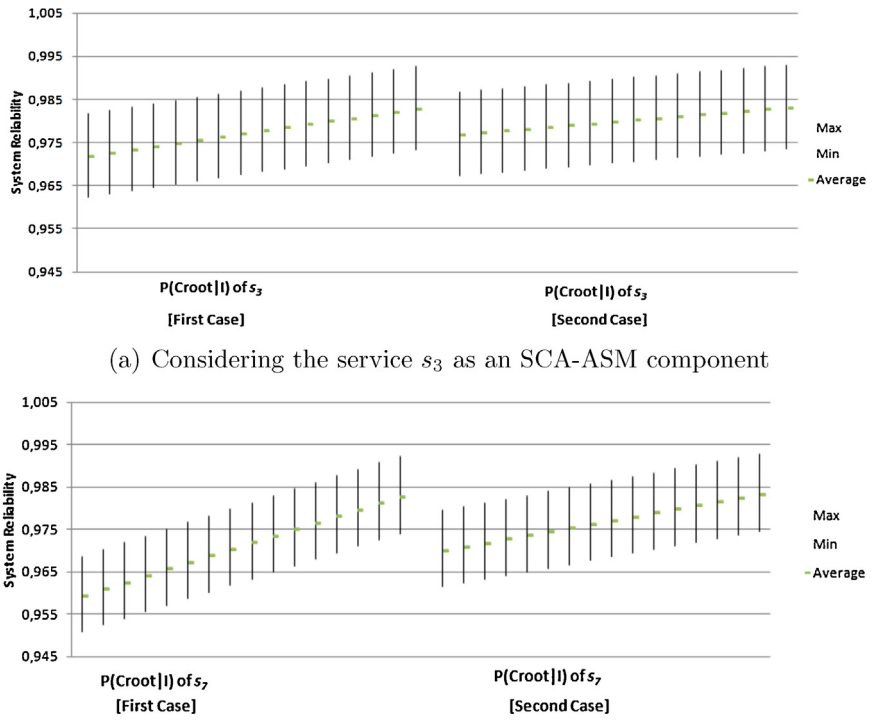

Fig. 20. System reliability obtained by embedding the reliability of an SCA-ASM component.

IU failures (see Eq. (10) in Section 4). We have predicted the system reliability for a set of value for the parameter $P(C_{root}|I)$. This latter spans from 0.08 to 0.005 by steps of −0.005. We have varied the parameter $P(C_{root}|I)$ under two different setting of the parameters μ_v. In the first case, we have assumed that μ_v is fixed to 0.7. In the second case, we have assumed that μ_v is fixed to 0.4. In both cases we have assumed the parameter ϕ_v fixed to $ln(0.9992)$.

Once fixed one μ_v parameter setting, each bar - corresponding to one $P(C_{root}|I)$ value - contains the higher, lower and the average value obtained for the system reliability. As expected, the lower (higher or average) system reliability increases while decreasing the $P(C_{root}|I)$ value. For example, in the first case (i.e., μ_v equals to 0.7), with $P(C_{root}|I)$ equals to 0.04, the average system reliability is about 0.97782, whereas if $P(C_{root}|I)$ decreases to 0.01, then the average reliability increases, and it is about 0.982177.

Similarly, in Fig. 20(b) we illustrate the results obtained by explicitly using for s_7 our SCA-ASM reliability model. The figure shows the system reliability while varying the reliability of s_7, assumed to be due to *IU* failures. A different behavior of the reliability model can be observed by fixing a value on the *x*-axis and observing the values on two figures while increasing the $P(C_{root}|I)$ value. This different behavior of the reliability model is mainly due to the fact that services s_3 and s_7 have a different probability to be invoked. In fact, the service s_3 is only used in the third external service, whereas the service s_7 is used in all external services (see Table 4).

These results highlight how our reliability model combines specific features of an SCA-ASM component into the overall reliability model and leads to a more precise (and less optimistic) reliability estimation.

7. Related work

In this paper, we propose a reliability prediction method for the SCA-ASM component model. Approaches for quality estimation based on a specific component model are also being defined in the state-of-the-art literature (see, for example, Ding and Jiang, 2009; Brosch et al., 2012 detailed below).

The work in Ding and Jiang (2009) presents a reliability computation for the SCA component model. It proposes a dynamic behavior model for specifying the components interface behavior by a notion of *port* and *port activities*. It defines failure behaviors of ports through the *Nonhomogeneous Poisson Process*(NHPP). Thus, the overall system reliability is computed on the reliability of port expressions.

The work in Brosch et al. (2012) proposes a reliability prediction method based on the Palladio Component Model (PCM), which offers a UML-like modeling notation. In particular, a tool is defined to automatically transform PCM models into Markov chains. On the contrary, our reliability approach relies on a unique component model (i.e. SCA-ASM), that is also used for static and dynamic modeling of software.

In this paper, we leverage the ASM formal method, and propose an approach for evaluating the reliability attribute for an architecture specified with SCA-ASM formal component model. Formal methods, such as Petri Net, Automata and Process Algebra, are also used by existing approaches for quality evaluation (see, for example, the work in Cortellessa et al., 2011 for performance modeling notations). Ad-hoc models (such as UML), used to describe the static and dynamic aspects of a software system, are typically transformed in formal quality models (such as Queueing Networks). Thus, a major problem of these approaches may reside in the distance between notations for modeling the system and notations for modeling qualities.

As far as the reliability is concerned, a quite extensive list of approaches can be found in literature (e.g., see survey Immonen and Niemelä, 2008). Most of these approaches (e.g., Cortellessa and Potena, 2007; Brosch et al., 2012; Ciancone et al., 2011) use notations based on UML sequence and deployment diagrams annotated with reliability properties, such as failure probabilities. Tools can transform such high-level models into analysis models (such as Markov models for state-based approaches), which then can be evaluated.

As an example, the Klapersuite (Ciancone et al., 2011) is a modeling framework (language, methodology, tools) for the predictive modeling and analysis of performance and reliability of component-based systems. It uses model transformations to automatically generate analysis models (queuing networks) out of an annotated UML design model.

As remarked in Ibrahim et al. (2011), the formal models typically focus only on the formal modeling of the service behavior. In Ibrahim et al. (2011), the formal specification of services with context-dependent contracts and their compositions is provided. Non-functional aspects are also taken into account, and the model checking technique is exploited to verify service properties w.r.t. the composition specification. The verification of functional and non-functional aspects based on a formal method is also performed in Aldini et al. (2010), where process algebraic techniques are used for architecture-level functional and performance analysis.

Regarding the novelty of our approach with respect to the state-of-art, we can remark the following points. (i) The work in Riccobene et al. (2012), which this paper is an extended version of, is one of the few papers proposing a reliability model for a component model, and (to the best of our knowledge) the first that relies on a formal and executable service-oriented component model, i.e. SCA-ASM, suitable for carrying out both architecture specification and its functional/non-functional properties analysis in an unified manner. In comparison with Riccobene et al. (2012), where we provided the bases of a reliability model for an SCA assembly of SCA-ASM components, this paper goes deeply into the details of the proposed method, makes precise those definition left abstract (as the usage profile model), makes complete the computation of several probabilistic parameters, and presents a deeper experimental evaluation. (ii) The framework we propose can facilitate the work of system architects and/or maintainers. In fact, our approach reduces the gap between notations for system modeling and notations for modeling qualities, so, therefore, strongly reduces the incompatibilities among models for quality analysis and system specification. (iii) The obtained experimental results support our intuition that the combination of the service orchestrator's reliability with those of the single service components may be a key factor for a trustworthy estimation of a software reliability.

8. Conclusions and future work

In this paper, we have defined a reliability model for service oriented applications developed using the SCA-ASM component model. In particular, we have presented a reliability model for an SCA assembly involving SCA-ASM components that embed the main service orchestration. We have also introduced a reliability model of an SCA-ASM component by considering failures specific to the nature of the ASMs.

We have pointed out the importance of the combination of the reliability prediction of the service orchestrator with those of other service components. To this extent, we have shown that our model leads to a more accurate estimation of the reliability. The conducted experimental results have also shown the effectiveness of the proposed approach with respect to BPEL-based approaches for service component reliability.

We intend to apply our approach to other examples to further study its scalability, and to compare its predicted reliability values with existing data of real life experiments. In order to support an automated application of the approach to large-sized problems, we are extending a prototype tool, based on the SCA-ASM tool-set (Riccobene et al., 2011; Riccobene and Scandurra, 2013) and on the SCA runtime platform Tuscany, to support the prediction method for a practical use. Besides, we intend to specialize our tool for supporting particular aspects of a quality model, such as the estimation of quality at runtime – model parameters estimation like reliability of components.

Other interesting research directions we intend to investigate concern the consideration of other QoS attributes (e.g., performance), and the introduction of dependency between failures of single components and failures specific of the nature of the ASMs.

Acknowledgements

We would like to thank the anonymous referees for their comments that helped to substantially improve the quality of the paper.

References

Abdelmoez, W., Nassar, D.M., Shereshevsky, M., Gradetsky, N., Gunnalan, R., Ammar, H.H., Yu, B., Mili, A., 2004. Error propagation in software architectures. In: IEEE International Symposium on Software Metrics, pp. 384–393.

Abdelmoez, W., Shereshevsky, M., Gunnalan, R., Ammar, H.H., Yu, B., Bogazzi, S., Korkmaz, M., Mili, A., 2005. Quantifying software architectures: an analysis of change propagation probabilities. In: AICCSA, IEEE Computer Society, p. 124.

Abreu, J., Fiadeiro, J.L., 2008. A coordination model for service-oriented interactions. In: COORDINATION. Lecture Notes in Computer Science, vol. 5052. Springer, pp. 1–16.

Aldini, A., Bernardo, M., Corradini, F., 2010. A Process Algebraic Approach to Software Architecture Design. Springer-Verlag London Limited, ISBN: 978-1-84800-222-7.

Alves, A., Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Sterling, D., König, V., Mehta, S., Thatte, D., van der Rijn, P., Yendluri, A., Yiu, May 2006. Web services business process execution language version 2.0, OASIS Committee Draft.

Arbab, F., 2004. Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14 (3), 329–366.

Arcaini, P., Gargantini, A., Riccobene, E., Scandurra, P., 2011. A model-driven process for engineering a toolset for a formal method. Journal of Software: Practice and Experience 41 (2), 155–166.

The ASMETA toolset website, 2011, http://asmeta.sf.net/

Börger, E., Stärk, R., 2003. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg.

Banti, F., Lapadula, A., Pugliese, R., Tiezzi, F., 2008. Specification and analysis of SOC systems using COWS: a finance case study. Electronic Notes in Theoretical Computer Science 235.

Brosch, F., Koziolek, H., Buhnova, B., Reussner, R., 2012. Architecture-based reliability prediction with the Palladio component model. IEEE Transactions on Software Engineering 38 (6), 1319–1339.

Brugali, D., Gherardi, L., Riccobene, E., Scandurra, P., 2011. Coordinated execution of heterogeneous service-oriented components by Abstract State Machines. In: FACS. Lecture Notes in Computer Science, vol. 7253. Springer.

Calinescu, R., Ghezzi, C., Kwiatkowska, M.Z., Mirandola, R., 2012. Self-adaptive software needs quantitative verification at runtime. Communications of the ACM 55 (9), 69–77.

Cardellini, V., Casalicchio, E., Grassi, V., Presti, F.L., 2007. Flow-based service selection for web service composition supporting multiple QoS classes. In: ICWS, IEEE Computer Society, pp. 743–750.

Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Presti, F.L., Mirandola, R., 2012. MOSES: a framework for QoS driven runtime adaptation of service-oriented systems. IEEE Transactions on Software Engineering 38 (5), 1138–1159.

Cardoso, J., Sheth, A., Miller, J., Arnold, J., Kochut, K., 2004. Quality of service for workflows and web service processes. Journal of Web Semantics 1 (3), 281–308.

Chandran, S.K., Dimov, A., Punnekkat, S., 2010. Modeling uncertainties in the estimation of software reliability. In: 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI), IEEE Computer Society, pp. 227–236.

Ciancone, A., Filieri, A., Drago, M.L., Mirandola, R., Grassi, V., 2011. KlaperSuite: an integrated model-driven environment for reliability and performance analysis of component-based systems. In: TOOLS (49). Lecture Notes in Computer Science, vol. 6705. Springer, pp. 99–114.

Cortellessa, V., Potena, P., 2007. Path-based error propagation analysis in composition of software services. In: Software Composition. Lecture Notes in Computer Science, vol. 4829. Springer, pp. 97–112.

Cortellessa, V., Marinelli, F., Potena, P., 2006. Automated selection of software components based on cost/reliability tradeoff. In: EWSA. Lecture Notes in Computer Science, vol. 4344. Springer, pp. 66–81.

Cortellessa, V., Marco, A.D., Inverardi, P., 2011. Model-Based Software Performance Analysis. Springer-Verlag, Berlin, Heidelberg, ISBN: 9783642136214.

Ding, Z., Jiang, M., 2009. Port based reliability computing for service composition. In: Proceedings of the 2009 IEEE International Conference on Services Computing, SCC'09, pp. 403–410.

EU project SENSORIA, www.sensoria-ist.eu/

Filieri, A., Ghezzi, C., Grassi, V., Mirandola, R., 2010. Reliability analysis of component-based systems with multiple failure modes. In: CBSE. Lecture Notes in Computer Science, vol. 6092. Springer, pp. 1–20.

Gargantini, A., Riccobene, E., 2001. ASM-based testing: coverage criteria and automatic test sequence. Journal of Universal Computer Science 7 (11), 1050–1067.
Gargantini, A., Riccobene, E., Scandurra, P., 2008. A metamodel-based language and a simulation engine for abstract state machines. Journal of Universal Computer Science 14 (12), 1949–1983.
Goseva-Popstojanova, K., Kamavaram, S., 2004. Software reliability estimation under uncertainty: generalization of the method of moments. In: HASE, IEEE Computer Society, pp. 209–218.
Goseva-Popstojanova, K., Trivedi, K.S., 2001. Architecture-based approach to reliability assessment of software systems. Performance Evaluation 45 (2–3), 179–204.
Grassi, V., 2004. Architecture-based reliability prediction for service-oriented computing. In: WADS. Lecture Notes in Computer Science, vol. 3549. Springer, pp. 279–299.
Ibrahim, N., Mohammad, M., Alagar, V.S., 2011. An architecture for managing and delivering trustworthy context-dependent services. In: IEEE SCC, pp. 737–738.
Immonen, A., Niemelä, E., 2008. Survey of reliability and availability prediction methods from the viewpoint of software architecture. Software and System Modeling 7 (1), 49–65.
Krka, I., Edwards, G., Cheung, L., Golubchik, L., Medvidovic, N., 2009. A comprehensive exploration of challenges in architecture-based reliability estimation. In: Architecting Dependable Systems VI. Lecture Notes in Computer Science, vol. 5835., pp. 202–227.
Mayer, P., Schroeder, A., Koch, N., 2008. A model-driven approach to service orchestration. In: IEEE SCC (2), pp. 533–536.
OSOA. Service Component Architecture (SCA). www.osoa.org
Riccobene, E., Scandurra, P., 2013. A formal framework for service modeling and prototyping. Formal Aspects of Computing, http://dx.doi.org/10.1007/s00165-013-0289-0 (on line first).
Riccobene, E., Scandurra, P., Albani, F.,2011. A modeling and executable language for designing and prototyping service-oriented applications. In: EUROMICRO-SEAA. IEEE, pp. 4–11.
Riccobene, E., Potena, P., Scandurra, P., 2012. Reliability prediction for service component architectures with the SCA-ASM component model. In: EUROMICRO-SEAA, IEEE Computer Society, pp. 125–132.
Smith, C.U., Williams, L.G., 2002. Performance Solutions: A Practical Guide to Creating Responsive. Scalable Software. Addison-Wesley, Reading.
Standard Glossary of Software Engineering Terminology, STD-729-1991 ANSI/IEEE, 1991.
Apache Tuscany [Online]. Available at: http://tuscany.apache.org/
van der Aalst, W.M.P., Pesic, M., 2006. DecSerFlow: towards a truly declarative service flow language. In: The Role of Business Processes in Service Oriented Architectures, Dagstuhl Seminar Proceedings, vol. 06291, Schloss Dagstuhl, Germany.
Voas, J.M., Miller, K.W., 1995. Software testability: the new verification. IEEE Software 12, 17–28.
Yacoub, S., Cukic, B., Ammar, H., 1999. Scenario-based reliability analysis of component-based software. In: 10th International Symposium on Software Reliability Engineering, 1999. Proceedings, pp. 22–31.
Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H., 2004. QoS-aware middleware for web services composition. IEEE Transactions on Software Engineering 30, 311–327.

Raffaela Mirandola is Associate Professor in the Dipartimento di Elettronica, Informazione e Bioingegneria at Politecnico di Milano. Raffaela's research interests are in the areas of performance and reliability modeling and analysis of software/hardware systems with special emphasis on: methods for the automatic generation of performance and reliability models for component-based and service-based systems, and methods to develop software that is dependable and can easily evolve, possibly self-adapting its behavior. She has published over 90 journal and conference articles on these topics. She served and is currently serving in the program committees of conferences in the research areas and she is a member of the Editorial Board of Journal of System and Software, Elsevier. She has been involved in several national and European research projects among which EU project CASCADAS (IST-027807), Q-ImPreSS (FP7-215013) and SMScom (IDEAS 227077).

Pasqualina Potena received the degree in Computer Science from the University of L'Aquila (Italy) and the PhD degree in Science from the University "G. D'Annunzio" Chieti e Pescara (Italy). Her research interests include: non functional-aspects (reliability, performance, cost, . . .), architecture-based solutions for self-adapting/evolving software systems, optimization models.

Elvinia Riccobene is associate professor in Computer Science at the University of Milan. She received her degree in Mathematics and her PhD degree in Applied Mathematics from the University of Catania (Italy). She holds visiting position at the Centre For High Assurance Computer System (Washington DC), at the Univ. of Bristol (UK), at the Univ. of Karlsruhe (Germany), at Florida University. Her research interests include formal methods and analyses techniques for software systems, integration between formal and semi-formal methods, model-driven engineering. She is the scientific coordinator of national research projects and she participated to various European and Italian research projects. She serves as member of the program committee of international conferences. She published several papers in International journals and in proceedings of international conferences.

Patrizia Scandurra obtained the Laurea degree (cum laude) in Computer Science in 2002 and a PhD in Computer Science in 2006, both from the University of Catania (Italy). After that, she got a Post-doc research position at the University of Milan (Italy). Since 2009 she is an assistant professor at the Engineering Department of the University of Bergamo. Her main research interests focused on the integration of formal and semi-formal modeling languages, formal methods and analysis (validation and verification) techniques, model-driven and component-based methodologies for the design and analysis of Pervasive Systems and Embedded Systems on Chip (SoC), Dynamic Software Architectures, and Service-Based Applications. She is a member of the Abstract State Machines (ASM) formal method community. She participated to different national and european research projects in the field of Software Engineering for modeling and analysis of pervasive systems, embedded systems, and robotic systems. She collaborated with companies such as STMicroelectronics, Opera21, and Atego, and with the universities of Milan, Pisa, Politecnico di Milano, Oxford, and Simula Research Laboratory in Oslo (Norway).