



Deposited via The University of York.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/75438/>

Version: Accepted Version

Proceedings Paper:

Johnson, Kenneth Harold Anthony, Calinescu, Radu Constantin and Kikuchi, Shinji (2013) An incremental verification framework for component-based software systems. In: CBSE '13 : Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering. 16th International ACM Sigsoft symposium on Component-based software engineering, 17-21 Jun 2013 ACM, CAN, pp. 33-42.

<https://doi.org/10.1145/2465449.2465456>

Reuse

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

An Incremental Verification Framework for Component-Based Software Systems

Kenneth Johnson
Dept. of Computer Science
University of York
York, YO10 5GH, UK
kenneth.johnson@york.ac.uk

Radu Calinescu
Dept. of Computer Science
University of York
York, YO10 5GH, UK
radu.calinescu@york.ac.uk

Shinji Kikuchi
Fujitsu Laboratories Limited
4-1-1 Kawasaki, Kanagawa
211-8588, Japan
skikuchi@jp.fujitsu.com

ABSTRACT

We present a tool-supported framework for the efficient re-verification of component-based software systems after changes such as additions, removals or modifications of components. The incremental verification engine at the core of our *INcremental VERification STRategy* (INVEST) framework uses high-level algebraic representations of component-based systems to identify and execute the minimal set of component-wise re-verification steps after a system change. The generality of the INVEST engine allows its integration with existing assume-guarantee verification paradigms. We illustrate this integration for an existing technique for the assume-guarantee verification of probabilistic systems. The resulting instance of the INVEST framework can reverify probabilistic safety properties of a cloud-deployed software system in a fraction of the time required by compositional assume-guarantee verification alone.

Categories and Subject Descriptors

D.2.4 [Software/Program Verification]: Model checking; D.2.11 [Software Architectures]: Languages

Keywords

Incremental Verification; Probabilistic Assume-Guarantee Verification; Domain-Specific Languages

1. INTRODUCTION

Formal verification techniques such as *model checking* have an excellent track record of successfully verifying critical hardware and software [8]. Given a finite state-transition model, model checking performs an exhaustive exploration of its state space in order to establish the satisfiability of a property expressed in some variant of temporal logic. The result is an irrefutable proof that the property is satisfied, or a counterexample comprising a sequence of state transitions that lead to its violation. The approach works particularly well for individual components and small systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CBSE'13, June 17–21, 2013, Vancouver, BC, Canada.
Copyright 2013 ACM 978-1-4503-2122-8/13/06 ...\$15.00.

For larger systems, standard model checking is rendered ineffective by high computation and memory overheads, and *compositional verification techniques* [1, 2, 7, 13, 15] are used instead. These techniques operate with significantly reduced overheads through verifying a large system one component at a time. Component interdependencies are taken into account by composing the models of individual components with *assumptions* that summarise the properties of other parts of the system they interact with.

In this paper, we are interested in a class of component-based software systems that are difficult to verify even using compositional techniques, namely systems whose components and structure change dynamically over time. Such systems are increasingly common, and include business-critical and safety-critical software from domains as diverse as health-care, transportation and finance [3, 18]. Verifying these software systems only at design time is insufficient [5, 10], and re-verifying them after each change that involves additions, removals or modifications of components is challenging [6]. To address this challenge, we propose an *INcremental VERification STRategy* (INVEST) that augments compositional verification with the ability to identify the minimal sequence of components requiring re-verification after each change.

As illustrated in Figure 1, the core functionality of the INVEST framework is provided by a *generic incremental verification engine*. This engine drives the re-verification of a component-based software system by using high-level algebraic representations of (a) the hierarchical structure of the system; and (b) a generic assume-guarantee verification paradigm. INVEST instances that integrate the engine with a specific *assume-guarantee model checker* can be further specialised through the use of a *domain-specific adaptor*, to support the incremental re-verification of software systems from that domain.

The main contributions of the paper are:

- An algebraically specified incremental verification engine

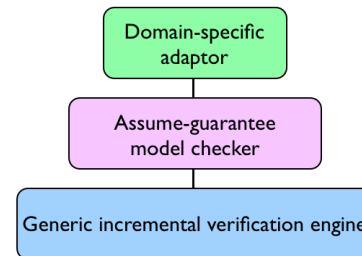


Figure 1: The layered architecture of the INVEST incremental verification framework

that extends, generalises and formalises the preliminary results we reported in [6]. Our approach is guided by the algebraic theory of data types [17], which designs *algebras* comprising system components and operations.

- The integration of the INVEST engine with the approach for the assume-guarantee verification of probabilistic systems introduced in [15]. The result of this integration is a framework for the incremental verification of probabilistic safety properties.
- The development of an INVEST adaptor that specialises the framework above for the incremental verification of probabilistic safety properties of multi-tier software applications deployed on cloud computing infrastructure.
- A prototype INVEST tool that integrates three elements: a Java implementation of our incremental verification engine, the widely used probabilistic model checker PRISM [14], and an adaptor for the incremental verification of component-based software systems deployed on cloud infrastructure.

The rest of the paper is organised as follows. Sections 2 to 4 present the three layers of the INVEST incremental verification framework. Section 5 describes our prototype implementation of the framework, and the experiments that we carried out to assess its effectiveness and scalability. We then discuss related work in Section 6, and finish with a brief summary and several concluding remarks in Section 7.

2. INCREMENTAL VERIFICATION

This section formulates the mathematical description of our incremental verification engine. The workflow of the engine is shown in Figure 2. It takes as input a high-level algebraic description S of a component-based system, and a function G that associates the system S and its components with properties from a property set P . The properties specified by G take values from a value set V , and need to be verified to “guarantee” compliance with the system requirements. The engine workflow comprises two stages:

1. In the *set-up stage*, S and G are provided to the engine for the first time, and standard compositional verification is used to form *assumptions* $A : P \rightarrow V^u$ that associate the properties in P with the appropriate values in V . Any properties from P that do not correspond to S or one of its components are by default mapped to an undefined value $u \in V^u = V \cup \{u\}$. The actions associated with this stage of the workflow are depicted using continuous lines in Figure 2.
2. In the *runtime stage*, the engine is notified whenever changes occur in the system. Changes such as component removals, additions or modifications correspond to substituting a *term* t from the algebraic description of the system S with another term t' . Accordingly, the engine handles a change $S[t/t']$ by updating the system description and using incremental reverification to update the assumptions $A : P \rightarrow V^u$. Figure 2 depicts the actions taken by the engine in this stage using dotted lines.

2.1 Specifying Component-Based Systems

A *component signature* Σ is a finite set C of component sorts, representing the hardware and software of a system. We use the symbol $C(\Sigma)$ to denote all components with the signature Σ and call $C(\Sigma)$ a *component class*. A *component algebra* is a component class $C(\Sigma)$ and a family $F =$

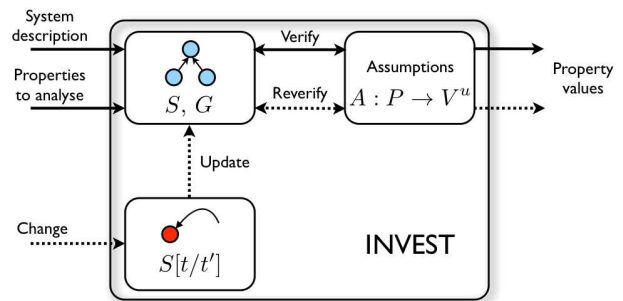


Figure 2: Incremental verification workflow

$\{f_1, \dots, f_m\}$ of operations on components. Given a component algebra we may choose any components and apply a sequence of operations. Such a sequence is represented as an algebraic expression called a component term or simply a *term*. Terms are defined recursively by the rules

$$t ::= c_1 \mid \dots \mid c_n \mid f_1(t_1, \dots, t_{n_1}) \mid \dots \mid f_m(t_1, \dots, t_{n_m}) \quad (1)$$

where $c_i \in C(\Sigma)$ and the t_j 's are terms.

Given a component algebra we can choose some (basic) components and apply a sequence of operations to create complex components, using the technique to describe the hierarchical structure of a component-based system. The terms used for this define the *calculations* carried out in the construction of the system description, and specify the *order* in which the operations on components are applied. Component algebras organise the operations we define, while the inductive properties of terms provide a natural data structure for defining an incremental verification process.

2.2 Component Model Checking

Let M be a set of models, P a set of properties and V a set of values. A model checker is an automated process in which the state space of a model $m \in M$ is exhaustively checked to verify the satisfiability of a property $p \in P$, returning a verification result $v \in V$. We suppose that the model checking process always terminates, potentially due to failure. If a failure does occur then a special unverified value u is returned. We extend the set V to include this value by defining $V^u = V \cup \{u\}$.

We consider verification paradigms for which the verification process may rely on values obtained from verifying other properties in P . We call total functions of the form $P \rightarrow V^u$ *assumptions*, and denote the set of all assumptions as $[P \rightarrow V^u]$. We define an assume-guarantee model checker as a function $mc : M \times P \times [P \rightarrow V^u] \rightarrow V^u$ such that

$$mc(m, p, A) = \text{the value obtained from verifying} \quad (2) \\ \text{property } p \text{ on model } m, \text{ assuming } A,$$

for a model $m \in M$, a property $p \in P$ and assumptions $A : P \rightarrow V^u \in [P \rightarrow V^u]$.

2.3 Set-up Stage: Compositional Verification

In order to carry out model checking on components in the class $C(\Sigma)$, each component is associated with a model and a property to verify by defining the total functions

$$m : C(\Sigma) \rightarrow M \text{ and } G : C(\Sigma) \rightarrow P,$$

called the *model* and (guaranteed) *property* functions respectively. The naive approach to model checking a component-based system $S \in C(\Sigma)$ comprising n components c_1, \dots, c_n

requires evaluating standard parallel composition [7, 8]

$$m(S) = m(c_1) \parallel \dots \parallel m(c_i) \parallel \dots \parallel m(c_n), \quad (3)$$

and obtaining a verification result from the model checker:

$$mc(m(S), G(S), A)$$

such that $A(p) = u$ for all $p \in P$, and $G(S)$ represents the system requirements of S . As parallel composition constructs a model of all possible interleavings of the components that make up the system, it can lead to extremely large model sizes. Despite recent advances improving symbolic model checking, this often results in an intractable verification task, even for medium-sized systems.

Compositional verification techniques are based on a divide and conquer approach that carries out component-wise verification of local properties in order to infer global properties of the system. Our approach of building up the system description S from basic components using a sequence of operations in the algebra $C(\Sigma)$ allows us to define a function $cv : C(\Sigma) \times [P \rightarrow V^u] \times [C(\Sigma) \rightarrow P] \rightarrow [P \rightarrow V^u]$ to carry out the compositional verification process such that

$$cv(S, A, G) = \text{updated assumptions obtained from} \\ \text{verifying properties } G : C(\Sigma) \rightarrow P \\ \text{for the model } m(S), \text{ assuming } A^u$$

where $A^u : P \rightarrow V^u$ is such that $A^u(p) = u$, for all $p \in P$.

We formulate a recursive definition of the function cv by induction on the structure of terms in $C(\Sigma)$.

Base case: The simplest case of compositional verification is on a component-based system comprising a single component. For each $c_1, \dots, c_n \in C(\Sigma)$ from (1) we define

$$cv(c_i, A, G)(p) = \begin{cases} mc(m(c_i), p, A) & \text{if } p = G(c_i), \\ A(p) & \text{otherwise.} \end{cases} \quad (4)$$

for all $p \in P$.

Structural induction on terms: For component terms t_1, \dots, t_m we consider the value of property $p \in P$ on the model associated with component $f(t_1, \dots, t_m)$, for an operation $f \in F$ of the form $f : C_1 \times \dots \times C_m \rightarrow C_0$. We have the following equation defined by two cases. First, if p is not a property associated with $f(t_1, \dots, t_m)$, i.e. $p \neq G(f(t_1, \dots, t_m))$ then no verification is required and

$$cv(f(t_1, \dots, t_m), A, G)(p) = A(p). \quad (5)$$

Otherwise, if $p = G(f(t_1, \dots, t_m))$ then

$$cv(f(t_1, \dots, t_m), A, G)(p) = mc(m(f(t_1, \dots, t_m)), p, A^m).$$

The assumption function A^m is formed by recursively applying compositional verification to each of t_1, \dots, t_m , i.e.,

$$A^1 = cv(t_1, A, G), \text{ and } A^{k+1} = cv(t_k, A^k, G). \quad (6)$$

2.4 Runtime Stage: Incremental Verification

In systems operating in dynamic environments changes such as the addition, removal or modification of components occur in rapid succession. Despite the significant improvements gained when applying compositional verification to large systems, it is inefficient to completely reverify the system after every such change.

To formulate an incremental verification procedure that performs only the necessary reverifications at runtime, we

suppose that the INVEST engine is notified of changes to the system. When a change notification arrives, the engine

1. updates the system representation to reflect the change,
2. constructs a sequence of the components affected by the change, and
3. performs the necessary reverification using results obtained in the previous (re)verification step, which may be the initial full verification of the system or the preceding incremental verification step.

Step 1: Component Change.

We model change in a system represented by the term S algebraically as a transformation on terms in the component class $C(\Sigma)$. For terms $t, t' \in C(\Sigma)$, we define the *change transformation*

$$S' = S[t/t'] \quad (7)$$

that results in the component term S' representing the changed system, with all instances of t in S substituted with t' .

Step 2: Reverification Sequences.

We formulate an incremental verification technique that involves the construction of a *reverification sequence*

$$\bar{\sigma} = (t_1, \dots, t_k)$$

comprising all S' sub-terms affected by (7). The sequence $\bar{\sigma}$ is constructed incrementally starting with $t_1 = t'$ and each t_i of $\bar{\sigma}$ satisfies the property t_i is a sub-term of t_{i+1} for $1 \leq i < k$, where $t_k = S'$. To ensure this property, we assume that the constants in the system's component term are unique (i.e. appear only once in the system specification).

A straightforward induction on the structure of $C(\Sigma)$ is used to construct a reverification sequence. Let Seq denote the set of sequences where $() \in Seq$ is the empty sequence, and \wedge is the concatenation operation on sequences. We define the function $\sigma : C(\Sigma) \times C(\Sigma) \rightarrow Seq$ as follows:

Base case: For a component $c_1, \dots, c_n \in C(\Sigma)$ we have

$$\sigma(c_i, t') = \begin{cases} (t') & \text{if } t' \equiv c_i, \\ () & \text{otherwise.} \end{cases}$$

Structural induction on terms: Let $f \in F$ be an operation of the form $f : C_1 \times \dots \times C_m \rightarrow C_0$. If t' is a sub-term of $f(t_1, \dots, t_m)$ then

$$\sigma(f(t_1, \dots, t_m), t') = \sigma(t_1, t') \wedge \dots \wedge \sigma(t_m, t') \wedge (f(t_1, \dots, t_m)).$$

Otherwise, $\sigma(f(t_1, \dots, t_m), t') = ()$.

Step 3: Incremental Reverification.

For this step, we suppose that compositional verification of the original system S has been carried out on the properties specified by $G : C(\Sigma) \rightarrow P$. In symbols,

$$A^{cv} = cv(S, A^u, G).$$

However, note that the results below also apply in the case when this compositional verification was followed by a number of incremental reverification steps.

We define the reverification function $\rho : Seq \times [P \rightarrow V^u] \times [C(\Sigma) \rightarrow P] \rightarrow [P \rightarrow V^u]$ such that $\rho(\sigma, A^{cv}, G) = A^r$ are the updated assumptions obtained from reverifying properties G for the components in the reverification sequence σ , assuming A^{cv} . We have the following definition:

Base cases: For the empty sequence we have $\rho(\epsilon, A^{cv}, G) = A^{cv}$. For the sequence (t_1) we have

$$\rho((t_1), A^{cv}, G)(p) = \begin{cases} mc(m(t_1), G(t_1), A^{cv}) & \text{if } p = G(t_1), \\ A^{cv}(p) & \text{otherwise.} \end{cases}$$

If $p = G(t_1)$ then the property p is model checked by mc on the component model $m(t_1)$, using the assumptions A^{cv} . The value resulting from the verification is returned and replaces the previous value $A(p)$ associated with p . Otherwise, the value associated with p remains unmodified.

Inductive step: Let $\bar{\sigma} = (t_i) \frown \alpha$, where α is a non-empty sequence; we define

$$\rho((t_i) \frown \alpha, A^{cv}, G) = \begin{cases} A' & \text{if } Stop(t_i), \\ \rho(\alpha, A', G) & \text{otherwise.} \end{cases} \quad (8)$$

where $A' = \rho((t_i), A, G)$ and

$$Stop(t_i) = compare(A'(G(t_i)), A^{cv}(G(t_i)))$$

provides a mechanism to terminate the reverification based on the satisfiability of the test

$$compare : V^u \times V^u \rightarrow \mathbb{B}. \quad (9)$$

Note that the value-equality function

$$compare_{=} (v^{new}, v^{old}) \equiv [v^{new} = v^{old}]$$

can always be used for this purpose. As we will show in Section 3, less demanding *compare* functions exist for specific assume-guarantee verification paradigms, enabling an earlier stop of the incremental verification process than *compare*₌. When such a “relaxed” *compare* function is used, the $A' = \rho(\sigma, A^{cv}, G)$ property-to-value mappings produced by the incremental verification is an under-approximation of the system properties. This is acceptable whenever the mappings A^{cv} satisfy the high-level system requirements, and *compare* is chosen such that an early stop in (8) occurs only after sufficient evidence was accumulated to conclude that the changed system is “equally or better able than before” at satisfying its requirements.

2.5 Summary

The primary elements of the incremental verification engine can be described succinctly as the tuple

$$e = (C(\Sigma), M, P, V, compare, mc). \quad (10)$$

The engine e is applied to a domain-specific problem by

- selecting a verification technique mc supporting assume-guarantee reasoning;
- formulating the models M , properties P and verification results V used by mc , and the *compare* function;
- defining components $C(\Sigma)$ and choosing operations relevant to the problem.

3. PROBABILISTIC ASSUME-GUARANTEEN VERIFICATION

In this section we illustrate the integration of our INVEST incremental verification engine with the assume-guarantee paradigm proposed in [15]. This integration is achieved by fixing the following elements of our algebraic engine (10):

- M is the set PA of probabilistic automata;
- P is the set of deterministic finite automata (DFA) specifying *probabilistic safety properties*;
- $V = [0, 1]$ is the set of probability values;
- the probability values are compared by *compare* : $V^u \times V^u \rightarrow \mathbb{B}$ from (9), defined as

$$compare(v_{new}, v_{old}) \equiv [v_{new} \geq v_{old}],$$

where inequality is extended to the special symbol u such that $v < u$, for all $v \in V$;

- $mc : M \times P \times [P \rightarrow V^u] \rightarrow [P \rightarrow V^u]$ is the assume-guarantee verification technique presented in [15].

Each of these elements from [15] is summarised below.

Modelling components as probabilistic automata

A probabilistic automaton is a tuple $(S, s_0, \alpha, \delta, L)$ comprising a finite set of states S corresponding to all the possible states of the real-world component being modelled, with the initial state denoted s_0 . The set α contains action symbols and $L : S \rightarrow 2^{AP}$ labels each state in S with atomic propositions from a set AP . The probabilistic transition function $\delta \subseteq S \times (\alpha \cup \{\tau\}) \times Dist$ is a function modelling the transitions between states, where $Dist$ is a set of discrete probability distributions over the states in S , and τ denotes a non-action symbol causing a self-looping transition (i.e. remaining in the current state). The possible transitions from a current state $s \in S$ to another state are given by the set $\delta(s) = \{(s, a, d) \mid (s, a, d) \in \delta\}$. Thus to determine the next state s' , an element from the set $\delta(s)$ is chosen non-deterministically, and the state s' is selected randomly according to the distribution $\delta(s)$.

Assume-guarantee verification of probabilistic systems

Probabilistic assume-guarantee reasoning extends compositional verification to probabilistic systems. Both the assumptions made about the system and their guarantees are probabilistic safety properties.

A probabilistic safety property-value pair $\langle \mathcal{E} \rangle_{\geq v}$ is specified by a probability bound v , and a DFA tuple

$$\mathcal{E} = (Q, \alpha, \delta, q_0, F)$$

with state set Q , alphabet α , transition function $\delta : Q \times \alpha \rightarrow Q$, initial state q_0 and accepting states $F \subseteq Q$. The finite words in the alphabet of \mathcal{E} express sequences of actions associated with prefixes of paths that do not satisfy the probabilistic safety property. The verification of a probabilistic safety property specified by $\mathcal{E} \in P$ on a model $m \in M$ determines the minimum probability $v \in V$ of not reaching an accepting state of \mathcal{E} . In symbols,

$$\langle \mathcal{E} \rangle_{\geq v} \iff mc(m, \mathcal{E}, A) = v$$

where mc carries out probabilistic assume-guarantee verification using assumptions $A : P \rightarrow V^u$ as described below.

Given the probabilistic safety properties \mathcal{A} and \mathcal{G} and their corresponding component models $m_1 \in M$ and $m_2 \in M$, respectively, mc is used to determine the probability v_2 such that $m_1 \parallel m_2$ satisfies $\langle \mathcal{G} \rangle_{\geq v_2}$ in two steps:

1. Starting with assumptions A^u , standard model checking is performed to obtain $v_1 = mc(\mathcal{A} \parallel m_1, \mathcal{A}, A^u)$. A new assumption function A is obtained such that

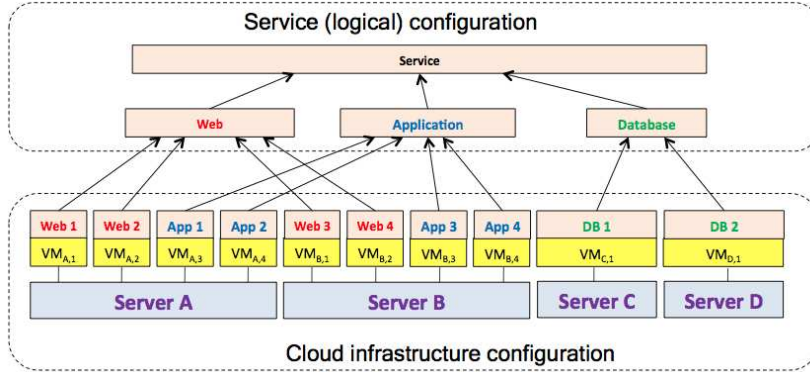


Figure 3: Three-tiered architecture of a cloud-deployed service

$A(\mathcal{A}) = v_1$, the minimum probability of not reaching the accepting states of \mathcal{A} , over all adversaries of m_1 .

- Using assumptions A , the probabilistic safety property \mathcal{G} is verified on the parallel composition of m_2 and DFAs \mathcal{A} and \mathcal{G} , to obtain

$$v_2 = mc(\mathcal{A} \parallel \mathcal{G} \parallel m_2, \mathcal{G}, A)$$

using *multi-objective model checking* [9]. A new assumption function A' is obtained where $A'(\mathcal{G}) = v_2$ is the minimum probability of not reaching the accepting states of \mathcal{G} , over all adversaries of m_2 and under the assumption $\langle \mathcal{A} \rangle_{\geq v_1}$.

These steps describe the probabilistic assume-guarantee rule introduced in [15], written in standard sequent notation as

$$\frac{\langle true \rangle m_1 \langle \mathcal{A} \rangle_{\geq v_1}, \langle \mathcal{A} \rangle_{\geq v_1} m_2 \langle \mathcal{G} \rangle_{\geq v_2}}{\langle true \rangle m_1 \parallel m_2 \langle \mathcal{G} \rangle_{\geq v_2}} \quad (11)$$

where $\langle true \rangle$ stands for the assumption function A^u .

4. DOMAIN-SPECIFIC ADAPTOR

To illustrate the application of the INVEST framework, we develop an adaptor specialising it for the analysis of probabilistic safety properties of multi-tier software services deployed on cloud computing infrastructure. This specialisation corresponds to fixing the element $C(\Sigma)$ of the INVEST algebraic engine (10), and enables administrators of multi-tier software to obtain answers to questions such as:

- what is the maximum probability of a service failing over a one-month time period?
- how will the probability of failure for my service be affected if one of its database instances is switched off to reflect a decrease in service workload?

A typical example of a software service that the adaptor developed in this section can handle is shown in Figure 3. This three-tiered service comprises *functions* for web, applications (app) and databases (db). Several instances of these three functions are run on different virtual machines (VMs) that are located on four physical servers. Note that although we chose this small example for illustration purposes, the framework is capable of handling systems that are significantly larger, as reflected by the experimental results presented later in the paper, in Section 5.

To apply the INVEST framework to multi-tiered services, we require a formalisation of the service and of the reliability properties to verify. A service deployed on the cloud consists

of components for hardware, software and logistics for managing the scaling up and down of the service. We specify an algebra for each sort of component and the operations we define will be used to construct a term representation of the whole service. In order to verify safety properties of the probabilistic automata associated with each component and with the service as a whole, we construct properties to be used as input for the INVEST engine.

4.1 Physical Server Components

Physical servers comprise a quantity of N_{mem} memory, N_{disk} hard disk and N_{cpu} CPU blocks that are initially operational, but are subject to failure over time. If a sufficient number of blocks fail, a signal is issued by the server's hardware failure detection and attempts are made to migrate all hosted virtual machines.

Let $C(\Sigma_{PS})$ denote the class of physical servers that represent the hardware infrastructure of the cloud data-centre. Each component in $C(\Sigma_{PS})$ is represented by a probabilistic automaton that models its operation over a specified time interval (e.g. one year). The model comprises a set of states corresponding to combinations of operational and failed blocks, with the initial state corresponding to the server state where all blocks are operational.

Figure 4 depicts the state transition diagram for the probabilistic automaton associated with a generic server. The sets of state transitions corresponding to the same action are annotated with the following action labels:

- `mem_op`, `disk_op` and `cpu_op`, denoting the operation of individual components;
- `server_detect`, `server_warn`, denoting correct operation of the failure mechanism;
- `server_up` and `server_down`, denoting the server operating correctly or failing by the end of the analysed time interval, respectively.

The states of the probabilistic automaton are labelled with atomic propositions expressing the amount of operational blocks in each state. We used illustrative values for the failure probabilities in order to aid readability, as the probabilities used in the experiments described in Section 5 (and which are based on results from [19, 21]) had too many decimal digits to fit in the diagram.

The physical servers from the case study are represented algebraically as component terms

$$serverA, serverB, serverC, serverD \in C(\Sigma_{PS}), \quad (12)$$

where each server comprises four CPUs, three hard disks and

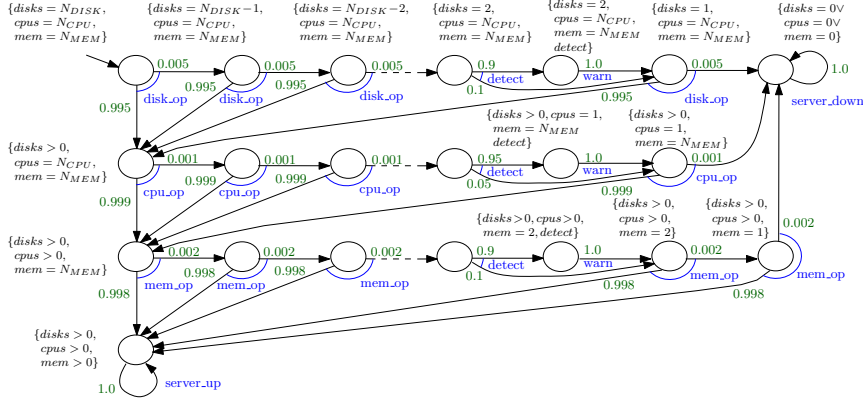


Figure 4: Probabilistic automaton modelling a physical server

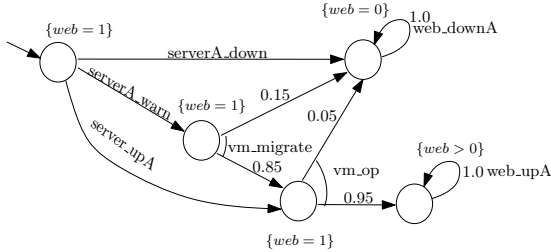


Figure 5: Model for component $deploy_{web}(serverA)$

eight memory blocks. The models $m(serverA)$, $m(serverB)$, $m(serverC)$ and $m(serverD)$ associated with the four servers are derived from the model in Figure 4 by adjusting the names of all probabilistic automaton actions to include the server identifier A, B, C or D, respectively (e.g., `server_down` is renamed `serverA_down` in the model $m(serverA)$).

4.2 Function Instance Components

Services are composed of several software components or *functions* $Func = \{func_1, \dots, func_k\}$ that deliver specific functionality required by the service. Each function is instantiated on one or more virtual machines hosted on servers across the data centre. The three-tier service in our case study, for instance, has $Func = \{web, app, db\}$ for its web, application and database functionality.

We extend the physical server algebra to form a new algebra $C(\Sigma_{FI})$ that comprises the class of ‘function instance’ components, generated by the operation

$$deploy_{func} : C(\Sigma_{PS}) \rightarrow C(\Sigma_{FI}),$$

for each $func \in Func$. For the web function used in the case study, the component term $deploy_{web}(serverA)$ represents the software function component of a single instance of the web function on *serverA*.

The probabilistic automaton $m(deploy_{web}(serverA))$ is depicted in Figure 5. The sets of state transitions labelled with the actions `serverA_up`, `serverA_down` and `serverA_warn` in this diagram correspond to the identically labelled actions from the model $m(serverA)$ of *serverA*. These transitions are not associated with probabilities since they depend on the behaviour of *serverA* (and will be specified by the assumptions used in the verification of the model). Note that this instance of the web function can fail in several ways: if *serverA* experiences a hardware failure; if the VM migration triggered by a warning fails; or if the VM fails (due to a software defect).

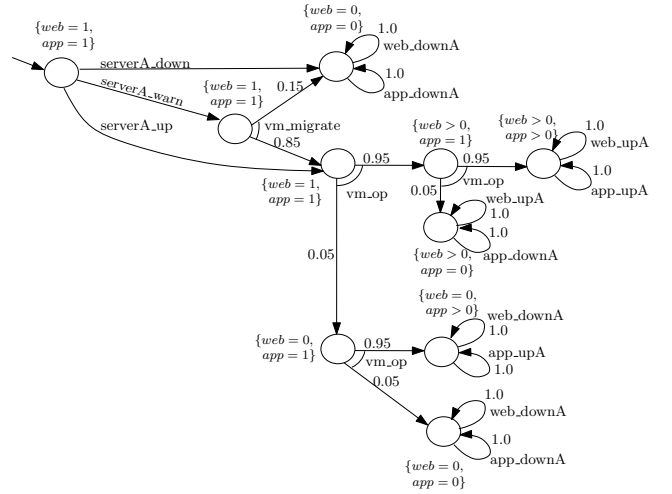


Figure 6: Model $m(add_{app}(deploy_{web}(serverA)))$

To specify the deployment of additional function instances onto an existing component in $C(\Sigma_{SF})$, we define the operation $add_{func} : C(\Sigma_{SF}) \rightarrow C(\Sigma_{SF})$. The component term $add_{app}(deploy_{web}(serverA))$ represents a composite function instance *webapp*, deployed on *serverA*, and the probabilistic automaton modelling it is shown in Figure 6.

Using the operations defined above, we can construct component terms for the *web*, *app* and *db* functions instances of our three-tier service as follows. First,

$$\begin{aligned} webappA &\equiv add_{app}^2(add_{web}(deploy_{web}(serverA))) \\ webappB &\equiv add_{app}^2(add_{web}(deploy_{web}(serverB))), \end{aligned} \quad (13)$$

represent components with two instances of web and app functions on *serverA* and *serverB*, respectively. Next,

$$\begin{aligned} dbC &\equiv deploy_{db}(serverC) \\ dbD &\equiv deploy_{db}(serverD) \end{aligned} \quad (14)$$

represent db function instances deployed on *serverC* and *serverD*, respectively.

4.3 Function Components

Services deployed in the cloud take advantage of its elastic nature, utilising data-centre resources as demand for the service requires. In this section, we develop a component class $C(\Sigma_F)$ of *software functions* that manages instances of identical functions, created for improved service availability and/or performance. We define the operation

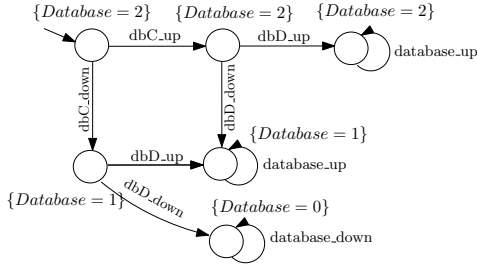


Figure 7: Model associated with the database in (15)

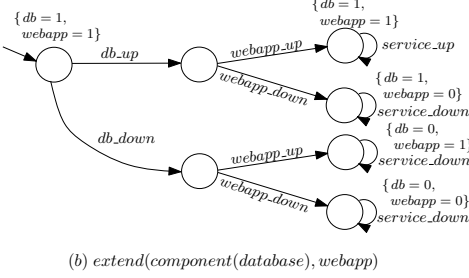


Figure 8: The probabilistic automaton $m(\text{service})$

function $func^n : C(\Sigma_{FI})^n \rightarrow C(\Sigma_F)$ that takes $n \geq 1$ function instances that provide identical functionality, and constructs a software function component in $C(\Sigma_F)$.

For example, the component term

$$\text{database} \equiv \text{function}_{db}(\text{dbC}, \text{dbD}) \quad (15)$$

specifies a new software function component for the case study function db, comprising two database instances dbC and dbD specified in (14). The resulting probabilistic automaton modelling (15) is depicted in Figure 7, and comprises states representing the operating status of the database instances dbC and dbD . The probabilities of the state transitions associated with the actions dbC_up , dbC_down and dbD_up , dbD_down are obtained (as assumptions) from the analysis the probabilistic automata $m(\text{dbC})$ and $m(\text{dbD})$. Similarly, we construct the component term

$$\text{webapp} \equiv \text{function}_{\text{webapp}}(\text{webappA}, \text{webappB}) \quad (16)$$

from the function instances webappA and webappB in (13).

4.4 Service Components

The component class $C(\Sigma_{Ser})$ of service components is generated by the operation $\text{svc}^n : C(\Sigma_F)^n \rightarrow C(\Sigma_{Ser})$ such that $\text{svc}^n(F_1, \dots, F_n)$ is a service whose functionality is implemented by the function components F_1, \dots, F_n in $C(\Sigma_F)$.

For function components $\text{database}, \text{webapp} \in C(\Sigma_F)$ defined in (15) and (16),

$$\text{service} \equiv \text{svc}^2(\text{webapp}, \text{database}) \quad (17)$$

is an algebraic specification of the service in our case study, and its model is depicted in Figure 8. The states correspond to the operational status of the database deployment of the service. Transitions are labeled with actions db_up , db_down and webapp_up , webapp_down that occur according to the probabilities obtained as assumptions in the verification of the database function and webapp function, respectively.

4.5 Service Probabilistic Safety Properties

This section constructs the properties $G_{\text{service}} : C(\Sigma_{Ser}) \rightarrow P$ mapping each term t representing a component of service from our case study to one or more DFAs. Since several components are associated with more than one property (i.e., DFA), P is the set of tuples of DFAs.

In the following, we adopt the shorthand notation $t \mapsto (\mathcal{E}_1, \dots, \mathcal{E}_n)$ to mean $G_{\text{service}}(t) = (\mathcal{E}_1, \dots, \mathcal{E}_n)$, for any component $t \in C(\Sigma_{Ser})$ and tuple $(\mathcal{E}_1, \dots, \mathcal{E}_n) \in P$.

Physical Servers

Figures 9(a)–(b) depict DFAs \mathcal{E}_1 and \mathcal{E}_2 formed from regular expressions over actions server_down and warn , specifying action sequences whose prefixes correspond to server failure and warning, respectively. For each term $\text{server} \in C(\Sigma_{PS})$ representing a case study server (12), we have $S \mapsto (\mathcal{E}_{1\text{server}}, \mathcal{E}_{2\text{server}})$ whose alphabet symbols are updated accordingly to server_warn and server_down .

Function Instances

We consider DFAs that have been generated specifically to determine the probability of a specific function instance failing, or all of them failing. The component $\text{webappA} \in C(\Sigma_{FI})$ is an instance of webapp , deployed on serverA and we identify the properties $\mathcal{E}_{3\text{webappA}}$, $\mathcal{E}_{4\text{webappA}}$ and $\mathcal{E}_{5\text{webappA}}$ whose DFAs are formed from the regular expressions

$$\begin{aligned} \text{reg}(\mathcal{E}_{3\text{webappA}}) &= (\text{app_upA}^+ \text{web_downA} \mid \\ &\quad \text{web_downA}^+ \text{app_upA})(\text{web_downA} \mid \text{app_upA})^* \\ \text{reg}(\mathcal{E}_{4\text{webappA}}) &= (\text{app_downA}^+ \text{web_upA} \mid \\ &\quad \text{web_upA}^+ \text{app_downA})(\text{web_up} \mid \text{app_downA})^* \\ \text{reg}(\mathcal{E}_{5\text{webappA}}) &= (\text{app_downA}^+ \text{web_downA} \mid \\ &\quad \text{web_downA}^+ \text{app_downA})(\text{web_downA} \mid \text{app_downA})^*. \end{aligned}$$

These regular expressions specify sequences of actions that correspond to web failure, app failure or the failure of both functions on serverA , and are synchronised during verification steps with actions from the model in Figure 6. Figure 9(c) shows the DFA for the first of these properties, and for a generic server (i.e., without the suffix 'A' appended to action names); the DFAs for the other two properties have the same structure, but are labelled with the appropriate actions from the other two regular expression.

Similarly, for component $\text{dbC} \in C(\Sigma_{FI})$, the DFA $\mathcal{E}_{6\text{dbC}}$ is specified by the regular expression $\text{reg}(\mathcal{E}_{6\text{db}}) = \text{db_downC}^+$ that corresponds to the failure of the db function instance on serverC . This DFA has the same structure as the DFA from Figure 9(a). We have the following mapping

$$\begin{aligned} \text{webappA} &\mapsto (\mathcal{E}_{3\text{webappA}}, \mathcal{E}_{4\text{webappA}}, \mathcal{E}_{5\text{webappA}}) \\ \text{webappB} &\mapsto (\mathcal{E}_{3\text{webappB}}, \mathcal{E}_{4\text{webappB}}, \mathcal{E}_{5\text{webappB}}) \\ \text{dbC} &\mapsto (\mathcal{E}_{6\text{dbC}}) \text{ and } \text{dbD} \mapsto (\mathcal{E}_{6\text{dbD}}). \end{aligned} \quad (18)$$

Functions

We consider DFAs specifying probabilistic safety properties that determine the probability of the failure of a specific software function deployed over one or more servers.

The DFAs \mathcal{E}_7 and \mathcal{E}_8 that we associate with functions db and webapp are formed from the regular expressions $\text{reg}(\mathcal{E}_7) = \text{database_down}^+$ and $\text{reg}(\mathcal{E}_8) = \text{webapp_down}^+$ corresponding to the failure of the db and webapp functions across all their respective instances. We map $\text{database} \mapsto$

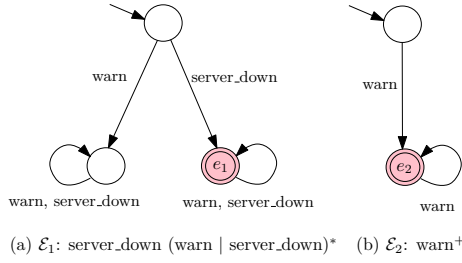


Figure 9: Probabilistic safety properties \mathcal{E}_1 , \mathcal{E}_2 and \mathcal{E}_3 from the case study

$\mathcal{E}_{7database}$, and $webapp \mapsto \mathcal{E}_{8webapp}$ for the function components $database, webapp \in C(\Sigma_F)$ in our case study.

Services

The probabilistic safety property associated with a service corresponds to the DFA specified by the regular expression $reg(\mathcal{E}_9) = service_down^+$. Its $service_down$ action is synchronised with the model $m(service)$ from Figure 8, and the verification of this property determines the minimum probability v_9 of the service not failing during the analysed time period. We have the mapping $service \mapsto \mathcal{E}_{9service}$.

5. IMPLEMENTATION AND VALIDATION

We developed a prototype INVEST incremental verification tool as an open-source Java application that is freely available from www-users.cs.york.ac.uk/~ken/invest. The core component of the INVEST tool is the generic class

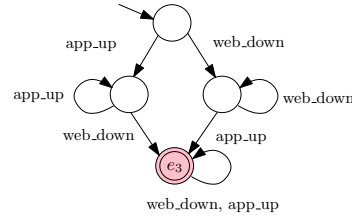
IncrementalEngine<M,P,V>

parameterised by three type variables M, P and V. These variables are abstract classes for models, properties and verification values, respectively, and the functions mc and $compare$ of the INVEST incremental engine from (10) are abstract methods of M and V , respectively.

For our case study, each component class presented in Section 4 was implemented as a Java class in INVEST. The classes comprise methods that implement the operations on probabilistic automata, represented in the state-based modelling language of the PRISM model checking tool [14].

The incremental verification engine implements the algorithms for compositional and incremental verification from Section 2 on a tree structure representation of the component-based system. Each node in this tree comprises a component model, properties to verify, and the available assumptions.

The INVEST tool was used to verify the component-based software system specified by the term $service$ in (17) in a range of experiments involving component additions, removals and modifications. In these experiments, the system and its components were first verified in full using compositional verification, to establish the A^{cv} property to value mappings for the initial system configuration shown in Figure 3; the results for this are shown in Table 1. Next, incremental verification was used to analyse the effects of series of change scenarios. A couple of these change scenarios are described below. For each of them, we present the steps taken by the engine when a change occurs and the new probability values resulting from incremental reverification. We are focusing on only two scenarios due to space constraints, but several other scenarios were tested successfully in our experiments, including physical server modifications and updates of the virtualisation software.



(c) $\mathcal{E}_3: (app_up^+web_down | web_down^+app_up) (web_down | app_up)^*$

Table 1: Compositional verification results from the set-up verification stage of the case study

Class	Model	Property value
$C(\Sigma_{PS})$	$serverA-D$	$v_1 = 1 - 1.2326E-6$
	$webappA-B$	$v_2 = 1 - 4.8332E-4$
$C(\Sigma_{FI})$	$databaseC-D$	$v_3 = 0.9975, v_4 = 0.9975$
	$database$	$v_5 = 1 - 5.5814E-5$
$C(\Sigma_F)$	$database$	$v_6 = 0.95$
	$webapp$	$v_7 = 0.9975$
$C(\Sigma_{Ser})$	$webapp$	$v_8 = 1 - 1.3571E-5$
	$service$	$v_9 = 0.9975$

Function Instance Removal.

Suppose that the physical server represented by the component $serverA$ has an unexpected hardware failure, and $webappA$ function instance becomes unavailable. The INVEST verification tool performs the following steps:

1. The term $service$ is updated using the substitution $service_1 \equiv service[webapp/function_{webapp}(webappB)]$.
2. The sequence $\bar{\sigma} = \sigma(function_{webapp}(webappB), service_1)$ of terms that may require verification is constructed.
3. Reverification is performed by calculating $A^1 = \rho(\bar{\sigma}, A^{cv}, G_{service})$ yielding the new assumption function A^1 . The reverification of $function_{webapp}(webappB)$ results in the new value $v_{8_{new}} = 0.9949$, and comparison with the old value $v_8 = 1 - 1.3571E-5$ yields

$$compare(v_{8_{new}}, v_8) = [0.9949 \geq 1 - 1.3571E-5] = false.$$

Since the stopping condition is not satisfied, reverification continues to the component $service_1$ resulting in the new verification value $v_{9_{new}} = 0.9924$. None of the other components is reverified.

Adding a Function Instance.

To improve the robustness of the service's database function, we suppose that the system administrator deploys (or considers deploying) a new instance of the db function on the physical server represented by the component $serverE$. The new db function instance is represented by the component term $dbE \equiv deploy_{db}(serverE)$. Once we extend the properties function $G_{service}$ with the mappings $serverE \mapsto (\mathcal{E}_{1E}, \mathcal{E}_{2E})$ and $dbE \mapsto (\mathcal{E}_{6E})$. The INVEST incremental verification engine carries out the following steps:

1. The term $service_1$ is updated by term substitution: $service_2 \equiv service_1[database/function(dbC, dbD, dbE)]$.
2. The reverification sequence $\bar{\sigma} = \sigma(serverE, dbE, service_2)$ is constructed.
3. Reverification involves calculating $A^2 = \rho(\bar{\sigma}, A^1, G_{service})$. Since the components $serverE, dbE$ have not been previ-

ously verified, they are included in the reverification process. The reverification of $function_{db}(dbC, dbD, dbE)$ results in the new verification value $v_{7_{new}} = 1-1.2535E-4$. Comparison with the old value $v_7 = 0.9975$ yields

$$compare(v_{7_{new}}, v_7) = [1-1.2535E-4 \geq 0.9975] = true,$$

and the incremental reverification process terminates.

Tool Performance.

To evaluate the performance of the INVEST tool, we ran a range of experiments in which we started from the system configuration in Figure 3, and then changed the system over a number of steps. The changes in these steps involved additions of webapp and db function instances (to assess the effect of increases in the system size) and modifications of existing components. The experiments were run on a standard 2.66 GHz Intel Core 2 Duo Macbook Pro computer with 8GB of memory, and all results (comparing the execution times of compositional and incremental verification) were averaged over multiple runs of each experiment. The results of three of these experiments are shown in Figure 10:

- In the experiment whose results are presented in the top-most graph, additional webapp instances similar to those from eq. (13) were added to the three-tier service from Figure 3, one instance at a time. After each such addition of a component, the properties of the upgraded system were reverified, using both compositional and incremental verification. The time taken by incremental verification was consistently lower than the time to reverify the system using compositional verification.
- The middle graph depicts the result of an experiment in which additional database instances with the form in eq. (14) were added to the original three-tier service from our case study. Like in the first experiment, incremental verification required significantly less time to reverify the changed system than compositional verification. The verification times were smaller than in the first experiment because a new database instance is associated with a single property to verify compared to three properties for a webapp instance, as shown in eq. (18).
- The last graph shows an experiment in which a disk block was removed from *serverC* in our case study, causing a degradation in the reliability of this server. The experiment was run for different system sizes, obtained through adding between 1 and 20 webapp and database instances to the original system. The incremental verification times were below 500ms for all system sizes in the experiment, while the compositional verification time grew rapidly with the system size, although the modification of a physical server was a localised system change. Even higher reductions in verification times were obtained when a physical server modification led to improved server reliability, in a separate experiment not shown in Figure 10 due to space constraints. In this case, the incremental verification time never exceeded 60ms, while the compositional verification completed in times similar to those from the last graph in Figure 10.

We expect that the use of the INVEST tool in a real-world environment will support a combination of scenarios similar to those from the experiments above, therefore providing significant reductions in the time required to reverify the properties of component-based systems after changes.

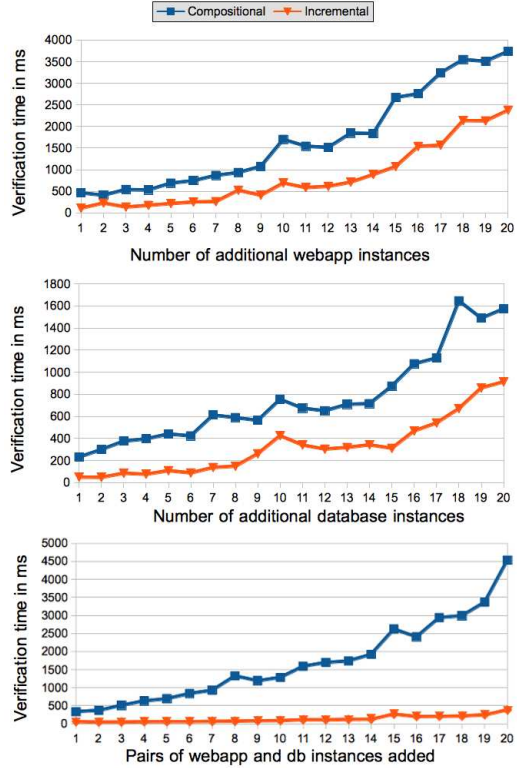


Figure 10: Experimental results

6. RELATED WORK

Although the concept of incremental analysis has been around for a number of decades, the need and possibility to adopt it in the area of formal verification have been identified and advocated only in recent years [5, 4, 10, 12]. As a result, only a few projects have explored the use of incremental verification so far, as described below.

The approaches reported in [16] and [11] exploit the strongly connected components of Markov decision processes to achieve incremental probabilistic verification after changes in state transition probabilities and model *guards* (i.e., predicates that describe permitted state transitions), respectively. These approaches support only a subset of the INVEST change scenarios, and are specific to a single type of models.

Ulusoy *et al.* [20] propose an incremental approach to synthesising Markovian models whose level of detail increases from one incremental step to the next. However, unlike INVEST, the analysis of these incrementally constructed models is performed using standard verification techniques.

Finally, the results presented in [10] use parameterised Markov chains whose property values are first computed algebraically, and then re-evaluated after each change in the model parameters. This technique handles well system modifications that correspond to changes in the transition probabilities of the analysed model. In contrast, INVEST also supports structural system changes, including component additions, removals and modifications that correspond to structural changes of individual component models.

7. CONCLUDING REMARKS

We presented the INVEST incremental verification framework for the reverification of component-based systems after

changes such as component additions, removals and modifications. The generic incremental verification engine at the core of the framework can augment existing assume-guarantee verification approaches with the ability to reverify component-based systems after changes efficiently, by reusing previous verification results whenever possible. To illustrate this characteristic of the INVEST engine, we described its integration with an existing approach for the assume-guarantee verification of probabilistic systems, and we “specialised” the result of this integration for the incremental verification of probabilistic safety properties of component-based software services deployed on cloud computing infrastructure. The effectiveness of the framework was assessed using a prototype implementation of all three layers of INVEST—the incremental verification engine, its integration with a probabilistic model checker, and the adaptor for the verification of cloud-deployed software services.

One advantage of our framework is its early completion of the incremental verification process. The reverification of the sequence of components affected by a change is stopped as soon as it is clear that the system as a whole is not any worse than prior to the change. To take full advantage of this capability, we are planning to extend INVEST with the ability to link property values to the overall system requirements, which can be regarded as predicates on the values of these properties. This will provide a basis for choosing a suitable definition for the *compare* function that underpins the early termination of an incremental verification step.

Another area of future work is the development of a domain-specific language (DSL) that will enable cloud datacentre administrators to use the INVEST tool to examine the reliability of cloud-deployed software services, and to explore “what if” scenarios associated with software and hardware upgrades and reconfigurations that they are planning, prior to implementing them. Without such a DSL, significant knowledge is required to take advantage of the capabilities of INVEST. Finally, we are investigating the integration of the INVEST incremental verification engine with other assume-guarantee verification paradigms, and the development of INVEST adaptors for other application domains. This integration will “fill in” the INVEST elements in Figure 1 with alternative building blocks. Another direction for future work is to explore the effectiveness of INVEST in handling property changes in an incremental manner.

Acknowledgements

This work was partly supported by the UK Engineering and Physical Sciences Research Council grant EP/H042644/1.

8. REFERENCES

- [1] S. Berezin, S. V. A. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *International Symposium on Compositionality: The Significant Difference*, pages 81–102. Springer, 1998.
- [2] C. Blundell, D. Giannakopoulou, and C. S. Pasareanu. Assume-guarantee testing. *ACM SIGSOFT Software Engineering Notes*, 31(2), 2006.
- [3] R. Calinescu. Emerging techniques for the engineering of self-adaptive high-integrity software. In J. Camara et al., editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *LNCS*, pages 297–310. Springer, 2013.
- [4] R. Calinescu et al. Dynamic QoS management and optimization in service-based systems. *IEEE Trans. Softw. Eng.*, 37:387–409, 2011.
- [5] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, 2012.
- [6] R. Calinescu, S. Kikuchi, and K. Johnson. Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In *Large-Scale Complex IT Systems*, volume 7539 of *LNCS*, pages 303–329. Springer, 2012.
- [7] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proc. 4th Intl. Symp. Logic in Computer Science*, pages 353–362, 1989.
- [8] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [9] K. Etessami, M. Kwiatkowska, M. Vardi, and M. Yannakakis. Multi-objective model checking of Markov decision processes. In *TACAS’07*, pages 50–65. Springer, 2007.
- [10] A. Filieri, C. Ghezzi, and G. Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Asp. Comput.*, 24(2):163–186, 2012.
- [11] V. Forejt et al. Incremental runtime verification of probabilistic systems. In *Runtime Verification*, volume 7687 of *LNCS*, pages 314–319. Springer, 2012.
- [12] C. Ghezzi. Evolution, adaptation, and the quest for incrementality. In *Large-Scale Complex IT Systems*, volume 7539 of *LNCS*, pages 369–379. Springer, 2012.
- [13] Y. Kesten and A. Pnueli. A compositional approach to CTL* verification. *Theor. Comput. Sci.*, 331(2–3):397–428, 2005.
- [14] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV’11*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
- [15] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Assume-guarantee verification for probabilistic systems. In *TACAS’10*, pages 23–37. Springer, 2010.
- [16] M. Kwiatkowska, D. Parker, and H. Qu. Incremental quantitative verification for Markov decision processes. In *DSN-PDS’11*, pages 359–370, 2011.
- [17] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 1, pages 189–368. Oxford University Press, 1992.
- [18] I. Sommerville et al. Large-scale complex IT systems. *Communications of the ACM*, 55(7):71–77, 2012.
- [19] K. Thomas. Solid state drives no better than others, survey says. http://www.pcworld.com/businesscenter/article/213442/solid_state_drives_no_better_than_others_survey_says.html.
- [20] A. Ulusoy, T. Wongpiromsarn, and C. Belta. Incremental control synthesis in probabilistic environments with temporal logic constraints. In *CDC’12*, pages 7658–7663, 2012.
- [21] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *SoCC ’10*, pages 193–204, New York, NY, USA, 2010. ACM.