Integration architecture synthesis for taming uncertainty in the Digital Space

Marco Autili, Vittorio Cortellessa, Davide Di Ruscio, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli

Università dell'Aquila
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
L'Aquila - Italy
{marco.autili,vittorio.cortellessa,davide.diruscio,paola.inverardi, patrizio.pelliccione,massimo.tivoli}@univaq.it

Abstract. The abundance of software that will be more and more available will promote the production of appropriate integration means (architectures, connectors, integration patterns). The produced software will need to be able to evolve, react and adapt quickly to a continuously changing environment, while guaranteeing dependability through (on-the-fly) validation. The strongest adversary to this view is the lack of information about the software, notably about its structure, behavior, and execution context. Despite the possibility to extract observational models from existing software, a producer will always operate with software artifacts that exhibit a degree of uncertainty in terms of their functional and non functional characteristics. Uncertainty can only be controlled by making it explicit and by using it to drive the production process itself. This calls for a production process that explores available software and assesses its degree of uncertainty in relation to the opportunistic goal G, assists the producer in creating the appropriate integration means towards G, and validates the quality of the integrated system with respect to the goal G and the current context. In this paper we discuss how goal-oriented software systems can be opportunistically created by integrating under uncertainty existing pieces of software.

1 Introduction

Increasingly, software applications will be produced following a production process paradigm that will be based on the reuse of non-proprietary software, often black-box and on software integrator systems that will ease the collaboration of existing software for the realization of new functionalities. The produced software will be inherently dynamic since it needs to operate in a continuously changing environment and must be able to quickly react and adapt to different types of changes, even unanticipated, while guaranteeing the dependability today's users expect.

This evidence promotes the use of an *experimental* approach, as opposed to a *creationistic* one, to the production of dependable[1] software. In fact, software development has been so far biased towards a *creationist view*: a producer is the owner of the artifact, and with the right tools she can supply any piece of information (interfaces, behaviors, contracts, etc.). The Digital Space promotes a different *experimental view*: the knowledge of a software artifact is limited to what can be observed of it. The more the observations will be powerful and costly the more the knowledge will be deep, but always with a certain degree of uncertainty. Indeed, there is a theoretical barrier that limits, in general, the power and the extent of observations.

The big challenge underlying this scenario is therefore to accept that this immense software resources availability corresponds to a lack of information about the software, notably about its behavior and on its execution context. A software producer will less and less know the precise behavior of a third party software service, nevertheless she will use it to build her own application. This means that the producer will operate in an environment in which the available services, and hence their related software artifacts (e.g., behavioral models, interface descriptions), exhibit a degree of uncertainty in terms of their functional and non functional characteristics (e.g., approximated behavioral models, incomplete interfaces, inaccuracy of performance parameters). We borrow Galbraith's definition of uncertainty, as taken from [35]: it "*defines uncertainty as the difference between the amount of information required to perform a task and the amount of information already possessed*". Indeed, in the software domain we see a flourishing of tools and methods to elicit approximated behavioral models of running systems. This problem recognized in the software engineering domain [23] is faced in many other computer science domain, e.g., exploratory search [45] and search computing [15], as well as software risk management [12], economics and other social domains [35]. In order to face this problem and provide a producer with a supporting framework to engineer the future software applications we envision a process that implements a radically new perspective.

In this paper we move some steps in the definition of EAGLE [4], an integrated model-based framework of theories, models, model-driven techniques, and tools to support the perpetual explore-integrate-validate production process of dependable software in the digital space, i.e., goal-oriented software systems that are opportunistically created by integrating under uncertainty existing software and that are dynamically evolving within a perpetually changing context. Specifically, we focus on the integration synthesis phase that aims at producing integration means to compose the explored software together in order to produce an application that satisfies the goal and that is able to tames uncertainty. The idea is that the integration solution compensates the lack of knowledge of the composed software by adding integration logic like connectors, mediators and adapters.

[1] We refer to the general notion of dependability, as defined by IFIP Working Group 10.4: "*the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers*".

The paper is structured as follows: Section 2 presents the state of the art and motivates the work. Section 3 recalls the Eagle approach while Section 4 explains the integration synthesis promoted by Eagle. Final remarks are discussed in Section 5, while the paper concludes in Section 6.

2 State-of-the-art overview

Eagle calls for uncertainty-aware and partial models. Uncertainty here corresponds to a measure, in a given metric system, of the incompleteness and inaccuracy of the models with respect to the goal G, which is due to the nature of the elicitation technique, its cost, and the operative context of the software system. Eagle systems opportunistically integrate pieces of software as available in a non-ideal world: this leads to accept incomplete information, hence accepting systems that represent the strictly necessary solution for satisfying the specified goal, possibly also in face of risks identification and prioritization. Thus, goal-oriented validation is another key aspect for Eagle. As discussed below, there exist many methods and techniques to account for uncertainty while developing software systems. All of them operate within different domains and consider uncertainty at different abstraction levels by also exploiting different software models. In this direction, one of the aims of Eagle is to combine/extend existing techniques and methods into a unified uncertainty-aware framework. Therefore, our state-of-the art overview is organized in several parts: Section 2.1 discusses approaches addressing the problem of deriving partial models from implemented systems. Section 2.2 presents the models@runtime approach and Section 2.3 concentrates on automatic connector synthesis to support software integration and coordination. Finally, Section 2.4 focuses on functional and non-functional Verification and Validation (V&V) under uncertainty.

2.1 Derivation of partial models

Many reverse engineering techniques have been applied to recover software architectural information from software systems. These techniques result in different structural models that describe approximations of the system internal structure [42]. Several approaches have recently addressed the problem of deriving partial behavioral models from implemented systems. In [8] we propose a method that combines synthesis and testing techniques in order to automatically derive the behavior protocol of a web-service out of its WSDL interface. In [22] the authors propose an approach to construct partial models for representing sets of alternatives and to use those alternatives for reasoning. In [44] the authors propose a synthesis technique that constructs partial behavioral models in the form of Model Transition Systems (MTS), a combination of safety properties and scenarios. In [32] the authors describe a technique to automatically generate behavioral models from (object-oriented) system execution traces. The work described in [25] aims to infer a formal specification of stateful black-box components that behave as data abstractions by observing their run-time behavior.

In [18] the authors propose tools and techniques to automatically derive models from running open source software systems in order to enable the simulation of their upgrades and to detect possible configuration inconsistencies.

2.2 Models@runtime

The models@runtime approach [9] seeks to extend the applicability of models produced in Model Driven Development (MDD) [39] approaches to the run-time environment. An example of design models application at run-time has been proposed by the PLASTIC project[2]. The PLASTIC development process [3, 2] relies on model-based solutions to build adaptable context-aware service-oriented applications. It encompasses methodologies and software tools to generate QoS models and adaptable application code from UML-based specifications. In this setting, opportunistic reuse of heterogeneous pieces of software, context awareness, run-time evolution, adaptiveness and uncertainty represent challenges that can be addressed by adopting a models@runtime approach [9]. Modeling techniques coupled with MDD capabilities, such as model transformation and code generation, provide viable means to enable system monitoring, model analysis and adaptation at run-time [27]. In [16] variability models are reused at run-time to support self-reconfiguration of systems when triggered by changes in the environment. In [34] run-time models of a system are used to reduce the number of configuration and reconfigurations to be considered when planning adaptations of the application. In [24] the use of configuration graphs is investigated as a means for monitoring and recording information about the system adaptations. As discussed in [31], meta-models allowing the definition of models where design- and run-time concepts are combined represent another key aspect for the creation and exploitation of effective run-time models. In the context of free and open source software systems, we use models@runtime to manage the upgrade of system configurations [18]. These approaches recognize the need to produce, manage and maintain software models all along the software's life time in order to assist the realization and validation of system's adaptations while the system is in execution.

2.3 Automatic connector synthesis to support software integration and coordination

The first approaches to connector synthesis appeared in the 90s in the control theory domain [38] and, thereafter, they have been revised to fit the domain of software (embedded) systems [1, 5]. The aim of these approaches is to automatically synthesize a controller that restricts the system behavior so as to satisfy a given specification. In [14, 37] LTSs are used to model the I/O behavior of components and automatically synthesize a set of constraints on the components' environment that allow deadlock avoidance. In [43] we show how to automatically derive either a centralized or distributed connector from a specification of

[2] FP6 IST EU PLASTIC, `http://www.ist-plastic.org/`

the components' interaction and of the requirements that the composed system must fulfill. However, these approaches do not take into account both possible run-time changes in the environment and non-functional requirements of the system to be integrated. The CONNECT project[3] overcomes these limitations promoting the development of automatic connector synthesis approaches that can be efficiently performed at run-time [29]. EAGLE aims at tackling the problem of automatically synthesizing integrators at run-time under uncertainty.

2.4 Functional and non-functional verification and validation under uncertainty

The idea of moving V&V activities at run-time [6] has been often realized by introducing monitoring activities both for functional and non-functional properties, and more recently by moving testing to on-line [7]. Uncertainty in EAGLE calls for compositional V&V techniques that permit to perform partial V&V (based on the information currently available) and to instrument the system so to be able to support on-line V&V. Many works have been proposed in compositional verification and, in particular, in assume-guarantee reasoning [17, 26, 19]. Bayesian models (such as Bayesian Networks [36]) can be considered as the stochastic counterpart of the assume-guarantee paradigm. In this direction, an example of bayesian approach for modeling the reliability of a software component-based system, given the reliability of its components, has been presented in [40]. More sophisticated stochastic models can be used to take into account uncertainty in non-functional validation processes. Hidden Markov Models (HMM) [20] are typically used to model systems that have Markovian characteristics in their behavior, but that also have some states (and transitions) for which only limited knowledge is available. Finally, theories [28] and techniques [10] for compositional approaches to testing have been investigated.

3 The EAGLE approach

The EAGLE approach promotes a novel production process (see Figure 1) that builds around three iterative phases explore-integrate-validate as follows [4]:

(i) **Explore:** explore available software services with the aim of extracting models as much complete as possible with respect to an opportunistic goal G. This means that, within the proposed software production process, we admit to deal with models that may exhibit a high degree of incompleteness, provided that they are accurate enough to satisfy user needs and preferences. For sake of validation, we will consider behavioral models annotated with quantitative non-functional parameters (e.g., Probabilistic Automata, UML+MARTE models, Queueing Networks, etc.);

[3] FP7 FET EU CONNECT, `http://connect-forever.eu/`.

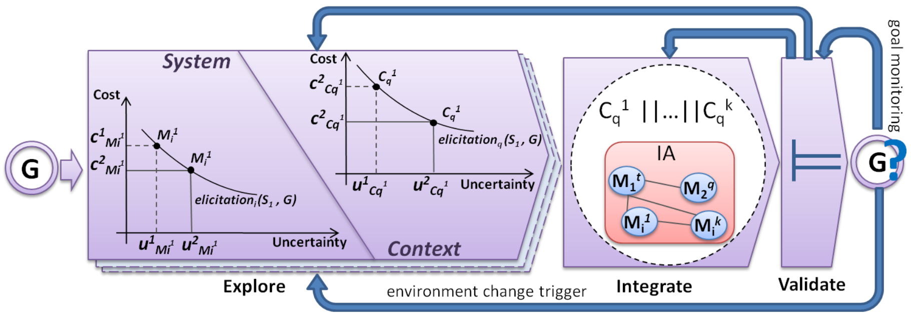

Fig. 1. Explore, Integrate, and Validate cycle

(ii) **Integrate:** assist the producer in creating the appropriate integration means to compose the explored software together in order to produce an application that satisfies G (e.g., from specific architectural integration patterns to solutions enforcing suitable architectural constraints). The integration solution can indeed compensate the lack of knowledge of the composed software by also adding integration logic like connectors, mediators and adapters.

(iii) **Validate:** dynamically validate the integrated system to assess its quality with respect to the goal G and the current context. This also requires to check whether a change in the goal or in the context occurs, so to seamlessly re-enact the explore-integrate-validate process to adapt to the change(s).

Feedbacks coming from validation and goal monitoring activities (see Figure 1) will instruct the process on whether proposing a new integration architecture (e.g., with the aim to act on the integration means, such as connectors, to avoid interactions that prevent the achievement of the goal), or reiterating the entire process to incrementally elicit more accurate software models (a specific lack of information in the considered models may lead to a meaningless validation). The explore-integrate-validate iteration is terminated once the validation step shows that the goal is achieved. Indeed, whenever changes in the monitored environment occur, the reiteration of the entire cycle might also be triggered (as new context may invalidate the goal).

In more details, according to Figure 1, if $S_1,\cdots,S_k$ are (with respect to the goal G) the candidate pieces of software that are being elicited by an explorative technique i, the result of an explorative phase, $elicitation_i(S_1,G),\ldots,$ $elicitation_i(S_k,G)$, is a set of models $M=\{M_i^1,\ldots,M_i^k\}$. Each model shall have associated its own accuracy, and hence its own metric for measuring the degree of uncertainty $u_{M_i^j}$. Moreover, each elicited model M_i^j has a cost $c_{M_i^j}$ that represents a quantitative measure of the effort to elicit M_i^j with an uncertainty degree $u_{M_i^j}$. The *Explore* box of Figure 1 shows a curve for the explorative technique i, that is able to elicit the model M_i^1 with different costs and uncertainty degree (along the curve). In general, a piece of software can have associated different models, as derived from different observations performed by different elicitation techniques. That is, the *Explore* box has a certain multiplicity (as represented by the dashed box boundaries) given by the multiplicity of the pieces of software under observation and of the explorative techniques. Similarly, different models of context $C=\{C_1,\ldots,C_n\}$ can be elicited and analogous definitions of uncertainty and cost metrics can be introduced for them.

4 Integration synthesis for taming uncertainty

As previously mentioned, we are primarily interested in extracting behavioral quantitative models of the software interaction protocols, and in modeling contexts together with their evolution. The elicited models can be incomplete and/or inaccurate with respect to the related software and the goal that the system has to achieve. The first refers to the behavioral modeling, i.e., less and/or more traces, the latter to the quantitative modeling, i.e., inaccurate probabilities and/or quantitative indices [33]. As anticipated in Section 2, in this context uncertainty corresponds to a measure, in a given metric system, of the incompleteness and inaccuracy of the models with respect to the goal G, which is due to the nature of the elicitation technique, its cost, and the operative context of the software system. Analogously to testing where the notion of coverage is pivotal to any metrics to assess the effectiveness of testing, reasoning about the quality of the elicited observational models needs similar notions. Indeed, in the Eagle scenario we are interested in developing systems by opportunistically integrating pieces of software and in assessing costs subsequent to choices, as in the "value-based" paradigm [11], so to achieve the goal most effectively. For the elicitation techniques, it shall therefore be possible to: (i) establish what portion of the goal specification can be fulfilled by the system under exploration, possibly under some assumptions on the environment; and (ii) select the suitable exploration techniques and establish a convenient strategy for their usage according to the cost of the elicitation process, as specified by the user preferences and needs.

To better explain how the integration synthesis promoted by Eagle tames uncertainty, let us introduce a hypothetical scenario of Eagle at work. Let us consider an e-commerce web service, *EcommerceWS*, with the aim of eliciting a behavioral model of it. The goal G is a combination of functional and non-functional properties, that can be informally expressed as follows: (i) to achieve a successful interaction among *EcommerceWS* and a client of it, i.e., to ensure that the client always progresses on buying items, (ii) to achieve a certain level of reliability of the whole system, where this attribute is given by the combination of the client and the web service reliabilities.

The **explore** step might use different techniques to elicit behavioral models of the software under exploration, e.g., from standard analysis techniques complemented with statistical inference to machine learning techniques [21, 13]. An elicited model M has a degree of uncertainty with respect to the system S and the goal G. In general, different models, each with its own degree of uncertainty, may exist. This is shown in the *Explore* box of Figure 1, where S can

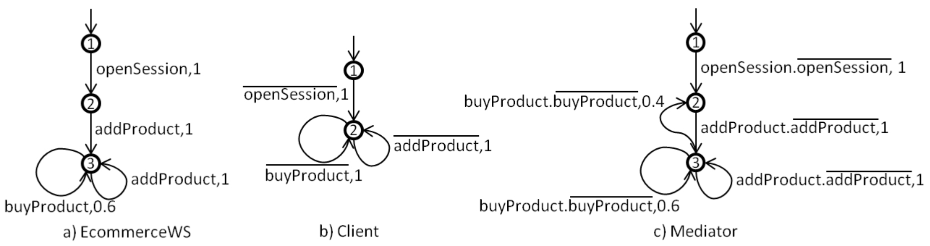
openSession,1
addProduct,1
addProduct,1
buyProduct,0.6
a) EcommerceWS
$\overline{\text{openSession}}$,1
$\overline{\text{addProduct}}$,1
$\overline{\text{buyProduct}}$,1
b) Client
openSession.$\overline{\text{openSession}}$, 1
buyProduct.$\overline{\text{buyProduct}}$,0.4
addProduct.$\overline{\text{addProduct}}$,1
addProduct.$\overline{\text{addProduct}}$,1
buyProduct.$\overline{\text{buyProduct}}$,0.6
c) Mediator

Fig. 2. *EcommerceWS* sample: explore and integration

have associated different models obtained through elicitation techniques with different costs and uncertainty degrees. Similarly, different models of context can be elicited and analogous definitions of uncertainty and cost metrics can be introduced for them. As a possible technique to be used in the **explore** step, we consider a version of the `StrawBerry` tool [8] that, for the EAGLE purposes, is enhanced to deal with the uncertainty degree of the elicited models.

Coming back to the example, clients of *EcommerceWS* can open a session, add a product to a shopping cart and buy items added to the cart. When an item is bought, it is removed from the cart. The operation used to buy a product, named *buyProduct*, is successfully concluded only if the shopping cart connected to the current session is not empty, an error will be raised otherwise. By taking as input the WSDL of *EcommerceWS*, the current version of `StrawBerry` produces a finite state automaton modeling the interaction protocol that a client has to follow in order to correctly interact with *EcommerceWS*. For the sake of the scenario, the enhanced version of `StrawBerry` shall produce the probabilistic automaton [41] for *EcommerceWS* in Figure 2.a. This automaton is potentially incomplete and the probabilities represent the uncertainty of the elicitation technique. Indeed, the operation *buyProduct* has a probability 0.6 to happen and to loop on state 3 (e.g., the case of successfully buying an item). The incompleteness of the model concerns the remaining cases in which *buyProduct* happens with a probability 0.4. In these cases, the model does not express what the behaviour of *EcommerceWS* may be, i.e., which states may be reached (e.g., when trying to buy an item from an empty cart). For instance, there might be other two *buyProduct* transitions from state 3, both with probability 0.2, going to state 1 and 2 respectively. The reason for this incompleteness of the model may depend on limits to the cost of the elicitation process as specified by user preferences and needs. Uncertainty on the behavior can also affect the estimate of reliability for sake of goal satisfaction, along with the uncertainty on the values of fundamental reliability parameters, such as the probability of failure of the *buyProduct* operation.

The **integration** step shall support the producer in creating the most effective (to the goal G) integration means that takes into account the uncertainty degree, and the associated cost, of each single elicited model. During the integration step, by reasoning on their elicited models and further accounting for the tradeoff between uncertainty degree and cost, the candidate pieces of software are selected. Then, an integration architecture IA is synthesized, possibly automatically (see the Integrate box of Figure 1). IA is synthesized by making assumptions on the uncertain behavior of the selected pieces of software, as well as on the uncertain reliability parameter values, in order to achieve the goal G. That is, the integrated system satisfies G only if the assumptions hold. As detailed later, the validation step is responsible to check such assumptions. Thus, IA plays a crucial role in influencing the overall uncertainty degree of the final integrated system S, as different IAs may result in different uncertainty degrees for S. By continuing our example, *EcommerceWS* and *Client* are the components that have been selected to build the integrated system. Actions denoted with the overbar in Figures 2 and 3 correspond to output actions, all the others correspond to inputs. The goal to be considered while producing the IA is that the integrated system, i.e., the one composed of *EcommerceWS*, *Client*, and the synthesized IA, always progresses on buying items with the required level of reliability. For instance, in Linear-time Temporal Logic, the functional part of goal G can be formally expressed as follows:

$$G =!(<> [\,](buyProduct))$$

whereas the non-functional one (in a simplified formulation) as follows:

$$Rel(Client) * Rel(EcommerceWS) >= targetrel$$

where $targetrel$ is the required level of reliability. In this example, IA assumes the form of a mediator (see Figure 2.c), which is an additional software entity that can be synthesized[4] [30]. It suitably mediates the interaction between *EcommerceWS* and *Client* in order to achieve G provided that some assumptions on the incomplete behavior and the reliability parameters of *EcommerceWS* hold. By referring to Figure 2.c, *Mediator* assumes that *EcommerceWS* reaches state 2 after a failure occurred while buying items. The transitions of the mediator model are labeled according to the following template:

<Client operation>.<*EcommerceWS* operation>, <probability to happen>

The mediator copes with the inherent uncertainty of the *EcommerceWS* model that concerns the case(s) in which *buyProduct* happens with a probability of 0.4. To this aim, *Mediator* assumes that *EcommerceWS* reaches state 2 after performing *buyProduct*,0.4. This assumption is reflected by the transition labeled *buyProduct.buyProduct*,0.4 in Figure 2.c. This transition is added during mediator synthesis to enforce the integrated system to perform *addProduct* once *buyProduct* has been executed with an empty shopping cart. For the non-functional part of the goal, assumptions are made on the reliabilities of service components, under a certain level of uncertainty.

[4] The CONNECT project (`http://connect-forever.eu/`, Grant agreement no. 231167) concerns the definition of theories and techniques to drop interoperability barriers by synthesizing on the fly the connectors via which networked systems communicate.

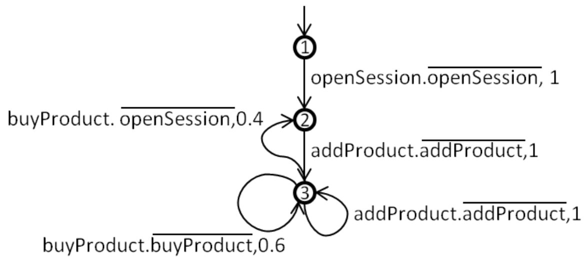
1
openSession.$\overline{\text{openSession}}$, 1
2
buyProduct. $\overline{\text{openSession}}$,0.4
addProduct.$\overline{\text{addProduct}}$,1
3
addProduct.$\overline{\text{addProduct}}$,1
buyProduct.$\overline{\text{buyProduct}}$,0.6

Fig. 3. *EcommerceWS* sample: validation

A **validation** step shall then assess the quality of the integrated system with respect to the assumptions made by IA. If the final assessment is not satisfying then the process shall iterate either to select different pieces of software, or to reduce the uncertainty degree of models (some already in place), or to modify the overall IA. For instance, back to the example, a new iteration of the explore-integrate-validate process is required when, upon validation, the above behavioral assumption made by *Mediator* does not hold. The new explore step will incrementally refine the elicited model by exploiting the feedbacks of the validation phase and the results of the previous explore step. In particular, the *EcommerceWS* model, of Figure 2.a, is refined by adding the transition *buyProduct*,0.4 from state 3 to 1. Consequently, a new mediator needs to be synthesized as shown in Figure 3. In general, although still incomplete with respect to the modeled software, the refined models might be accurate enough to achieve the functional part of G. In particular, the new mediator detects the failure of *buyProduct* and, by exploiting authentication information previously stored, simulates an access of Client to *EcommerceWS* by performing *openSession.* The non-functional validation of the integrated system can also report an uncertain result, such as "the system reliability is within an interval of 10% around $targetrel$", for example due to incomplete information about the reliability of some software components. In this case, either the process is reiterated, or (if feasible) the goal can be loosened and the integration acceptable.

5 Discussion

In this section we discuss some aspects that merit to be further investigated.

– **Metrics to quantify/qualify the uncertainty** - The metrics adopted to reason on uncertainty should be different depending on the sources of uncertainty they refer to. Furthermore, in some cases uncertainty cannot be quantified due to the source domain it stems from, thus it has to be qualified in non-ambiguous terms. Hence, uncertainty can be quantified/qualified in different ways. In the following we propose some examples:
  1. The uncertainty can be originated by a set of available alternatives (such as static, dynamic, or deployment alternatives) when more than one

   alternative can be suitable with respect to the goal G. In this case the uncertainty can be quantified (i) either with a probability assigned to each suitable alternative, when knowledge is sufficient to generate a set of values that sum up to 1, (ii) or with a non-stochastic metric that represents the level of preference/priority associated to each suitable alternative.
2. When uncertainty stems from functional or non-functional parameters of the model (e.g., maximum multiplicity of a component, resource demand of a service) the uncertainty can be quantified with intervals that bind the suitable values of these parameters.
3. In some other cases, e.g., in the case of a macro-component with an internal structure not completely known, uncertainty can be qualified through elements of the design. In these cases it could be appropriate to define/use partial specification modalities.

  Since different metrics can be used to measure the uncertainty of a piece of software, each one related to a specific aspect (e.g., behavior, reliability, performance, etc.), they have to co-exist in a coherent metric system. Therefore, such system should also contain relations and dependencies among these different metrics.
- **Tradeoffs between different metrics -** As anticipated, the uncertainty of a system is measured by means of a metric system. Then, this calls for tradeoffs between the different functional or non-functional aspects to be considered, each related to a suitable metric of uncertainty. In other words, within a suitable space of solutions determined by all uncertainties still in place, often a designer has to take decisions that decrease uncertainty in one direction whereas increase uncertainty in other directions. For example, in order to increase the reliability a higher number of (replicated and differently designed) components are put in place, whereas this choice, at the same time, increases the uncertainty about the resource demand of this system because many more components' demands have to be estimated.
- **Uncertainty estimation -** The explore phase of EAGLE produces a model of a software artifact specialized to represent some aspects. Quite often this model is defined under uncertainty that is associated to one or more metrics. Now a question raises, that is: *how to estimate the value of uncertainty metrics?* Referring to the example in Section 4, `StrawBerry` makes use of testing to extract the model, thus in this case the metric of uncertainty can be estimated by considering the number of positive and negative tests that have been performed. The knowledge of the designer can help during the explore phase since she can be aware, for instance, that a piece of software requires an amount of CPU that is in the range of $[x, y]$, with x and y belonging to the real numbers, event though no exact value is known.
- **Uncertainty of composed systems -** While building a system composed of several components or subsystems, the uncertainty metric system might be derived out of the uncertainty metric systems of the single components or subsystems. This calls for mechanisms to create a new metric system out of existing ones. Thus, relations and dependencies of the component

metric systems have to be exploited, and/or new relations and dependencies among metrics must be inferred. Let us now focus on a single metric. The measure of uncertainty related to this metric for a composed system can be calculated by suitably combining the metrics of uncertainty associated to the single components or subsystems. Thus, a suitable operator must be aptly adopted. In the example described in Section 4, the composition is simply performed by multiplying the probabilities associated to *Client* and *ECommerceWS*.

6 Conclusion

Eagle proposes a model-based framework for supporting the perpetual explore-integrate-validate cycle that will be realized by exploiting model-driven techniques. This integrated framework will support the engineering of goal-oriented software systems that are opportunistically created by integrating, under uncertainty, existing software and that are dynamically evolving within a perpetually changing context.

Reaching this goal requires to put at work different expertises and skills together, hence asking for a multi-domain research and development work on functional and non functional system modeling, verification and validation, model-driven development, context-aware programming, connector synthesis, and techniques for run-time monitoring and reconfiguration. As a by-product of this approach we expect that Eagle results should be exploitable in a multitude of contexts both research-wise and industrial-wise.

Acknowledgment

This work is supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreements: number 257178 (project CHOReOS - Large Scale Choreographies for the Future Internet - `www.choreos.eu`), and number 231167 (project CONNECT - Emergent Connectors for Eternal Software Intensive Networked Systems - `http://connect-forever.eu/`).

References

1. E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, 1995.
2. M. Autili, P. Benedetto, and P. Inverardi. Context-aware adaptive services: The plastic approach. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 124–139, Berlin, Heidelberg, 2009. Springer-Verlag.
3. M. Autili, L. Berardinelli, V. Cortellessa, A. Di Marco, D. Di Ruscio, P. Inverardi, and M. Tivoli. A development process for self-adapting service oriented applications. In *Proc. of ICSOC '07*, pages 442–448, 2007.

4. M. Autili, V. Cortellessa, D. Di Ruscio, P. Inverardi, P. Pelliccione, and M. Tivoli. Eagle: engineering software in the ubiquitous globe by leveraging uncertainty. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 488–491, New York, NY, USA, 2011. ACM.
5. C. Baier, M. Grer, M. Leucker, B. Bollig, and F. Ciesinski. Controller synthesis for probabilistic systems (extended abstract). In *IFIP TCS 2004*, volume 155. 2004.
6. A. Bertolino, G. Angelis, L. Frantzen, and A. Polini. Software engineering. chapter The PLASTIC Framework and Tools for Testing Service-Oriented Applications, pages 106–139. Springer-Verlag, Berlin, Heidelberg, 2009.
7. A. Bertolino, G. De Angelis, and A. Polini. (role)CAST : A Framework for On-line Service Testing. In *Proc. of WEBIST'11*, 2011.
8. A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proc of ESEC/FSE '09*, 2009.
9. G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42:22–27, 2009.
10. C. Blundell, D. Giannakopoulou, and C. S. Păsăreanu. Assume-guarantee testing. *Softw. Eng. Notes*, 31, 2005.
11. B. Boehm. Value-based software engineering: reinventing. *SIGSOFT Softw. Eng. Notes*, 28:3–, March 2003.
12. B. W. Boehm. Software risk management: Principles and practices. *IEEE Softw.*, 8:32–41, January 1991.
13. R. Calinescu, K. Johnson, and Y. Rafiq. Using observation ageing to improve markovian model learning in qos engineering. In *ICPE*, pages 505–510, 2011.
14. C. Canal, P. Poizat, and G. Salaün. Synchronizing behavioural mismatch in software composition. In *FMOODS*, pages 63–77, 2006.
15. S. Ceri, D. Braga, F. Corcoglioniti, M. Grossniklaus, and S. Vadacca. Search computing challenges and directions. In *Proc of ICOODB'10*, pages 1–5, 2010.
16. C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42:37–43, October 2009.
17. J. M. Cobleigh, D. Giannakopoulou, and C. Pasareanu. Learning Assumptions for Compositional Verification. In *Proc. of TACAS 2003*, number 2619 in LNCS, 2003.
18. R. Di Cosmo, D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli. Supporting Software Evolution in Component-Based FOSS Systems. *Science of Computer Programming*, 76(12), 2011.
19. J. Dingel. Computer-Assisted Assume/Guarantee Reasoning with VeriSoft. In *Proc. of ICSE2003*.
20. Y. Ephraim and N. Merhav. Hidden markov processes. *IEEE Transactions on Information Theory*, 48:1518–1569.
21. M. D. Ernst and J. H. Perkins. Learning from executions: Dynamic analysis for software engineering and program understanding, Tutorial at ASE 2005.
22. M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.
23. D. Garlan. Software engineering in an uncertain world. In *Proc. of FSE/SDP'10*, pages 125–128, 2010.
24. J. C. Georgas, A. van der Hoek, and R. N. Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer*, 42:52–60, 2009.
25. C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *Proc. of ICSE '09*, pages 430–440, 2009.

26. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. In *ASE journal, 12(3): 297-320*, 2005.
27. H. J. Goldsby and B. H. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *Proc. of MoDELS '08*, pages 568–583, 2008.
28. D. Hamlet. *Composing Software Components: A Software-testing Perspective.* Springer Publishing Company, Incorporated, 1st edition, 2010.
29. P. Inverardi, V. Issarny, and R. Spalazzese. A theory of mediators for eternal connectors. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II*, ISoLA'10, pages 236–250, Berlin, Heidelberg, 2010. Springer-Verlag.
30. P. Inverardi, R. Spalazzese, and M. Tivoli. Application-layer connector synthesis. In *SFM*, pages 148–190, 2011.
31. G. Lehmann, M. Blumendorf, F. Trollmann, and S. Albayrak. Meta-modeling runtime models. In J. Dingel and A. Solberg, editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science*, pages 209–223. Springer Berlin / Heidelberg, 2011.
32. D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proc. of ICSE '08*, pages 501–510, 2008.
33. K. Mishra and K. Trivedi. Uncertainty propagation through software dependability models. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 80 –89, 29 2011-dec. 2 2011.
34. B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42:44–51, 2009.
35. J. Mula, R. Poler, J. Garcia-Sabater, and F. Lario. Models for production planning under uncertainty: A review. *IJPE*, 103(1):271–285, 2006.
36. M. Neil, N. Fenton, and M. Tailor. Using bayesian networks to model expected and unexpected operational losses. *Risk Analysis*, 25(4):963–972, 2005.
37. R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: two faces of the same coin. In *Proc. of ICCAD '02*, pages 132–139, 2002.
38. P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81 –98, jan 1989.
39. D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.
40. H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj. A bayesian approach to reliability prediction and assessment of component based systems. In *Proc. of ISSRE '01*, 2001.
41. M. Stoelinga. An introduction to probabilistic automata. *Bulletin of the European Association for Theoretical Computer Science*, 78:176–198, 2002.
42. C. Stringfellow, C. D. Amory, D. Potnuri, A. Andrews, and M. Georg. Comparison of software architecture reverse engineering methods. *Information and Software Technology*, 48(7):484–497, July 2006.
43. M. Tivoli and P. Inverardi. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program.*, 71(3):181–212, May 2008.
44. S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Softw. Eng.*, 35:384–406, May 2009.
45. R. W. White and R. A. Roth. *Exploratory Search: Beyond the Query-Response Paradigm.* Synthesis Lect. on ICRS. Morgan & Claypool Publishers, 2009.