Article:

Calinescu, Radu Constantin orcid.org/0000-0002-2678-9260, Weyns, Danny, Gerasimou, Simos orcid.org/0000-0002-2706-5272 et al. (3 more authors) (2018) Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases. IEEE Transactions on Software Engineering. 8008800. pp. 1039-1069. ISSN 0098-5589

https://doi.org/10.1109/TSE.2017.2738640

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSE.2017.2738640

Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases

Radu Calinescu, Danny Weyns, Simos Gerasimou, M. Usman Iftikhar, Ibrahim Habli, and Tim Kelly

Abstract—Building on concepts drawn from control theory, *self-adaptive software* handles environmental and internal uncertainties by dynamically adjusting its architecture and parameters in response to events such as workload changes and component failures. Self-adaptive software is increasingly expected to meet strict functional and non-functional requirements in applications from areas as diverse as manufacturing, healthcare and finance. To address this need, we introduce a methodology for the systematic ENgineering of TRUstworthy Self-adaptive sofTware (ENTRUST). ENTRUST uses a combination of (1) design-time and runtime modelling and verification, and (2) industry-adopted assurance processes to develop trustworthy self-adaptive software *and* assurance cases arguing the suitability of the software for its intended application. To evaluate the effectiveness of our methodology, we present a tool-supported instance of ENTRUST and its use to develop proof-of-concept self-adaptive software for embedded and service-based systems from the oceanic monitoring and e-finance domains, respectively. The experimental results show that ENTRUST can be used to engineer self-adaptive software systems in different application domains and to generate dynamic assurance cases for these systems.

Index Terms—Self-adaptive software systems, software engineering methodology, assurance evidence, assurance cases.

1 Introduction

Software systems are regularly used in applications characterised by uncertain environments, evolving requirements and unexpected failures. The correct operation of these applications depends on the ability of software to adapt to change, through the dynamic reconfiguration of its parameters or architecture. When events such as variations in workload, changes in the required throughput or component failures are observed, alternative adaptation options are analysed, and a suitable new software configuration may be selected and applied.

As software adaptation is often too complex or too costly to be performed by human operators, its automation has been the subject of intense research. Using concepts borrowed from the control of discrete-event systems [91], this research proposes the extension of software systems with *closed-loop control*. As shown in Fig. 1, the paradigm involves using an external software *controller* to monitor the system and to adapt its architecture or configuration after environmental and internal changes. Inspired by the autonomic computing manifesto [67], [73] and by pioneering work on self-adaptive software [71], [86], this research has been very successful. Over the past decade, numerous research projects proposed architectures [54], [76], [124] and frameworks [15], [43], [109], [123] for the engineering of *self-adaptive systems*. Extensive surveys of this research and its applications are available in [68], [89], [95].

In this paper, we are concerned with the use of self-adaptive software in systems with strict functional and non-functional requirements. A growing number of systems are expected to fit this description in the near future. Service-based telehealth systems are envisaged to use self-adaptation to cope with service failures and workload variations [15], [44], [120], avoiding harm to patients. Autonomous robots used in applications ranging from manufacturing [40], [58] to oceanic monitoring [19], [55] will need to rely on self-adaptive software for completing their missions safely and effectively, without damage to, or loss of, expensive equipment. Employing self-adaptive software in these applications is very challenging, as it requires assurances about the correct operation of the software in scenarios affected by uncertainty.

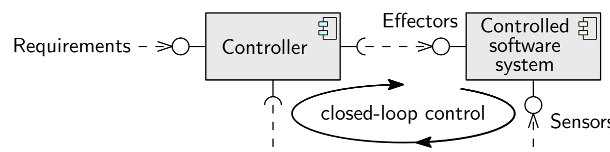

Fig. 1. Closed-loop control is used to automate software adaptation

Assurance has become a major concern for self-adaptive software only recently [25], [30], [36], [37]. Accordingly, the research in the area is limited, and often confined to providing evidence that individual aspects of the self-adaptive software are correct (e.g. the software platform used to execute the controller, the controller functions, or the runtime adaptation decisions). However, such evidence is only one component of the established industry process for the assurance of software-based systems [11], [81], [111]. In real-world applications, assuring a software system requires the provision of an *assurance case*, which standards such as [112] define as

> *"a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment"*.

- *R. Calinescu, S. Gerasimou, I. Habli and T. Kelly are with the Department of Computer Science at the University of York, UK.*
- *D. Weyns is with the Department of Computer Science of the Katholieke Universiteit Leuven, Belgium.*
- *M. U. Iftikhar is with the Department of Computer Science at Linnaeus University, Sweden.*

Our work addresses this discrepancy between the state of practice and the current research on assurances for self-adaptive software. To this end, we introduce a generic methodology for the joint development of trustworthy self-adaptive software systems *and* their associated assurance cases. Our methodology for the ENgineering of TRUstworthy Self-adaptive sofTware (ENTRUST) is underpinned by a combination of (1) design-time and runtime modelling and verification, and (2) an industry-adopted standard for the formalisation of assurance arguments [60], [102].

ENTRUST uses design-time modelling, verification and synthesis of assurance evidence for the control aspects of a self-adaptive system that are engineered before the system is deployed. These design-time activities support the initial controller enactment and the generation of a partial assurance case for the self-adaptive system. The dynamic selection of a system configuration (i.e., architecture and parameters) during the initial deployment and after internal and environmental changes involves further modelling and verification, and the synthesis of the additional assurance evidence required to complete the assurance case. These activities are fully automated and carried out at runtime.

The ENTRUST methodology is not prescriptive about the modelling, verification and assurance evidence generation methods used in its design-time and runtime stages. This generality exploits the fact that the body of evidence underpinning an assurance case can combine verification evidence from activities including formal verification, testing and simulation. As such, ENTRUST assurance cases can use assurance evidence obtained through a combination of testing, simulation and formal verification, at both design time and runtime.

ENTRUST supports the systematic engineering and assurance of self-adaptive systems. In line with other research on self-adaptive systems (see e.g. [95], [122]), we assume that the controlled software system from Fig. 1 already exists, and we focus on its enhancement with self-adaptation capabilities through the addition of a high-level monitor-analyse-plan-execute (MAPE) control loop. The components of the controlled software system may already support low-level, real-time adaptation to localised changes. For instance, the self-adaptive embedded system used in one of our case studies is a controlled unmanned vehicle that employs built-in low-level control to maintain the speed selected by its high-level ENTRUST controller. Mature approaches from the areas of robust control of discrete-event systems (e.g. [80], [91], [107], [126]) and real-time systems (e.g. [77], [82]) already exist for the engineering of such low-level control. Thus, real-time control is outside the scope of ENTRUST.

Likewise, established assurance processes are available for the non-self-adaptive aspects of software systems (e.g. [10], [11], [62], [65], [94]). We do not duplicate this work. Using these processes to construct assurance arguments for the correct design, development and operation of the controlled software system, and for the derivation, validity, completeness and formalisation of the requirements from Fig. 1 is outside the scope of our paper. Thus, ENTRUST focuses on the correct engineering of the controller and on the correct operation of self-adaptive system, assuming that the controlled system and its requirements are both correct.

The main contributions of our paper are:

1) The first end-to-end methodology for (a) engineering self-adaptive software systems with assurance evidence for the controller platform, its functions and the adaptation decisions; and (b) devising assurance cases whose assurance arguments bring together this evidence.
2) A novel assurance argument pattern for self-adaptive systems, expressed in the Goal Structuring Notation (GSN) standard [60] that is widely used for assurance case development in industry [102].
3) An instantiation of our methodology whose stages are supported by the established modelling and verification tools UPPAAL [7] and PRISM [79].

These contributions include four significant extensions of complementary results from our previously separate strands of work on developing formally verified control loops [69], runtime probabilistic model checking [20] and dynamic assurance cases [38]. First, the instantiation of the ENTRUST methodology is based on a formally verifiable controller architecture where the controller from [69] was extended to use probabilistic model checking at runtime [20]. Second, we introduce a set of generic properties that ENTRUST controllers must satisfy. Third, we extend our preliminary work from [38] with a realisation of the principles of dynamic assurance case continuity, updatability, proactivity, automation and formality that we suggested in [38]. Fourth, we devise the first assurance argument pattern for self-adaptive systems. In addition, we integrate these extended building blocks into a complete methodology for the engineering of self-adaptive systems.

To ensure the generality of ENTRUST, these contributions are evaluated using two case studies with different characteristics (e.g. types of system, requirements and adaptation actions) and belonging to different application domains (i.e. oceanic monitoring and exchange trade). We chose for these case studies systems that have been used to evaluate related software engineering research [19], [55], [57], [98], as these systems are already known to the research community – one of them as an "exemplar" for the evaluation of new approaches to engineering self-adaptive systems [56].

The remainder of the paper is organised as follows. In Section 2, we provide background information on assurance cases, GSN and assurance argument patterns. Section 3 introduces the self-adaptive systems used in our case studies, and Section 4 describes the generic ENTRUST methodology. Sections 5 and 6 present the tool-supported ENTRUST instance and its use to develop the self-adaptive systems from the two case studies, respectively. Section 7 presents our evaluation results, which show that the methodology can be used for the effective engineering of self-adaptive systems from different domains and for the generation of dynamic assurance cases for these systems. In Section 8, we overview the existing approaches to providing assurances for self-adaptive software systems, and we compare them to ENTRUST. Finally, Section 9 concludes the paper with a discussion and a summary of future work directions.

2 Preliminaries

This section provides background information on assurance cases, introducing the assurance-related terminology and

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSE.2017.2738640

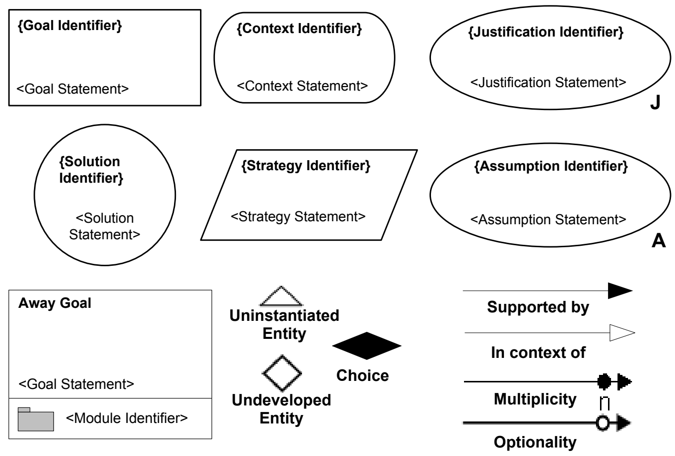

Fig. 2. Core GSN elements

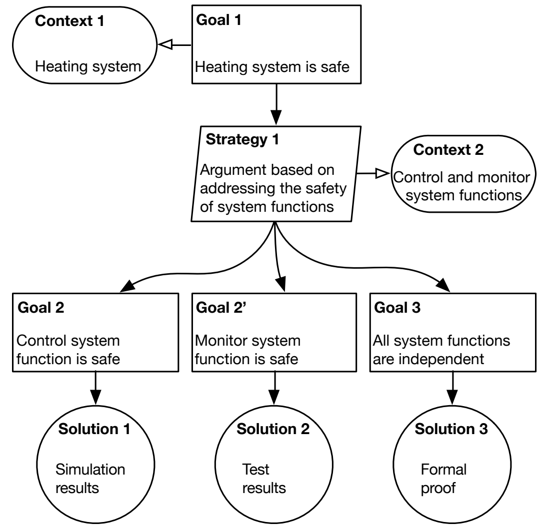

Fig. 3. Example of a GSN assurance argument

concepts used in the rest of the paper. We start by defining assurance cases and their components in Section 2.1. Next, we introduce a commonly used notation for the specification of assurance cases in Section 2.2. Finally, we introduce the concept of an assurance argument pattern in Section 2.3.

2.1 Assurance Cases

An *assurance case*[1] is a report that supports a specific *claim* about the requirements of a system [10]. As an example, the assurance case in [85] provides documented assurance that the "*implementation and operation of North European Functional Airspace Block (NEFAB) is acceptably safe according to ICAO, EC and EUROCONTROL safety requirements.*" The documented assurance within an assurance case comprises (1) *evidence* and (2) structured *arguments* that link the evidence to the claim [10], possibly through intermediate claims.

Assurance cases are becoming mandatory for software systems used in safety-critical and mission-critical applications [11], [81], [111]. They are used in domains ranging from nuclear energy [113] and medical devices [115] to air traffic control [45] and defence [112]. A growing number of assurance cases from these and other domains are openly available (e.g., [85], [114]).

The development of assurance cases comprises processes carried out at all stages of the system life cycle [111]. Requirements analysis evidence and design evidence demonstrate that system reliability, safety, maintainability, etc. are considered in the early stages of the life cycle. Implementation, validation and verification evidence are then generated as the system is developed. Finally, evidence collected at runtime is used to update assurance cases during system maintenance.

As aptly described in [111], the assurance case must be "*a living, cradle-to-grave document.*" This is particularly true for self-adaptive software systems. For these systems, existing evidence needs to be continuously combined with new *adaptation evidence*, i.e., evidence that the system will continue to operate safely after self-adaptation activities.

1. Assurance cases developed for safety-critical systems are also called *safety cases*. In this work, we are concerned with any self-adaptive software systems that must meet strict requirements, and therefore we talk about assurance cases and assurance arguments.

2.2 Goal Structuring Notation

The assurance cases for self-adaptive systems introduced later in the paper are devised in the *Goal Structuring Notation* (GSN) [72], a community standard [60] widely used for assurance case development in industry [102]. The main GSN elements (Fig. 2) can be used to construct an argument by showing how an assurance claim (represented in GSN by a *goal*) is broken down into sub-claims (also represented by GSN *goals*), until eventually it can be supported by GSN *solutions* (i.e., assurance evidence from verification, testing, etc.). *Strategies* are used to partition the argument and describe the nature of the inference that exists between a goal and its supporting goal(s). The rationale (*assumptions* and *justifications*) for individual elements of the argument can be captured, along with the *context* (e.g. to describe the operational environment) in which the claims are stated.

In a GSN diagram, claims are linked to strategies, subclaims and ultimately to solutions using '*supported by*' connectives, which are rendered as lines with a solid arrowhead and declare inferential or evidential relationships. 'Supported by' connectives may be decorated with their multiplicity or marked as optional. The '*in context of*' connective, rendered as a line with a hollow arrowhead, declares a contextual relationship between a goal or strategy on the one hand and a context, assumption or justification on the other hand.

Large or complex sections of the assurance argument can be organised into modules by means of GSN *away goals* referenced in the main argument and defined separately. Finally, GSN entities can be marked as *uninstantiated* to indicate that they are placeholders that need to be replaced with a concrete instantiation, and GSN goals can be marked as *undeveloped* to indicate that they need to be further developed into sub-goals, strategies and solutions.

As an example, Fig. 3 shows a simple GSN assurance argument for the software part of a heating system. Its root goal (**Goal 1**) claims that the system is safe at all times. This claim is partitioned into sub-claims using a strategy (**Strategy 1**) that addresses the safety of the two system functions (i.e. control and monitoring) separately through

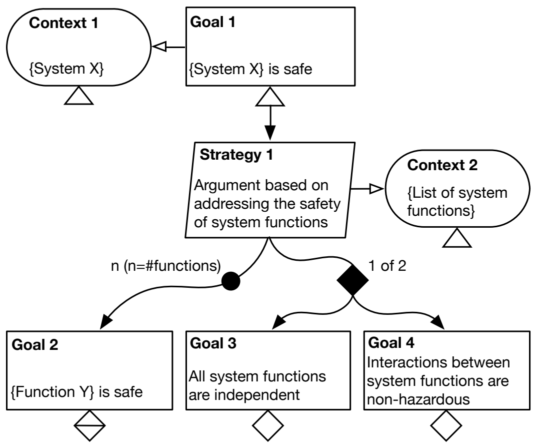

Fig. 4. Example of a GSN assurance argument pattern

sub-claims **Goal 2** (for the control system) and **Goal 2′** (for the monitor system), and includes sub-claim **Goal 3** that the two functions are independent. The three sub-claims are supported by three solutions comprising assurance evidence from simulation, testing and formal proof, respectively.

2.3 Assurance Argument Patterns

To reduce the significant effort required to develop assurance cases, in our previous work on software assurance [62], [64] we collaborated to the creation of a catalog of reusable GSN *assurance argument patterns* [63]. Each pattern considers the contribution made by the software to system hazards for a particular class of systems and scenarios. The GSN elements of a pattern that are generic to the entire class are fully developed and instantiated, whereas the entities that are specific to each system and scenario within the class are left undeveloped and/or uninstantiated.

As an example, Fig. 4 depicts an assurance argument pattern that is instantiated by the GSN assurance argument from Fig. 3. The elements surrounded by curly brackets '{' and '}' in the pattern must be *instantiated* for each assurance argument based on the pattern, as further indicated by the triangular 'uninstantiated' symbol under the GSN entities that contain them. **Goal 2** is marked with both this 'uninstantiated' symbol (because it contains elements in curly brackets) and a diamond-shaped 'undeveloped' symbol (because, like for the 'choice' sub-claims **Goal 3** and **Goal 4**, additional GSN entities must be added underneath to complete the assurance argument); the two symbols are rendered overlapping under **Goal 2**.

In this paper, we devise a new assurance argument pattern, which is applicable to self-adaptive software systems.

3 Self-adaptive Systems Used in the Case Studies

This section introduces the self-adaptive software systems from the two case studies used to illustrate and evaluate our methodology. To assess the generality of ENTRUST, we chose different types of systems from different domains. The first system, introduced in Section 3.1, is an embedded unmanned underwater vehicle (UUV) system from the oceanic monitoring domain. The second system, presented in Section 3.2, is a service-based system from the foreign exchange (FX) trade domain. Table 1 lists several additional characteristics that differ significantly between the two systems. These characteristics include the types of requirements, sensor data and adaptation actions of the systems, and the types of models whose verification underpins their self-adaptation decisions.

TABLE 1
Comparison of systems used to assess the generality of ENTRUST

	UUV	**FX**
Type	embedded system	service-based system
Domain	oceanic monitoring	exchange trade
Requirements	throughput, resource use, cost, safety	reliability, response time, cost, safety
Sensor data	UUV sensor measurement rate	service response time and reliability
Adaptation actions	switch sensors on/off, change speed	change service instance
Uncertainty modelling	continuous-time stochastic model of UUV sensors	discrete-time stochastic model of system

3.1 Unmanned Underwater Vehicle (UUV) System

The self-adaptive UUV embedded system is adapted from [55]. UUVs are increasingly used in a wide range of oceanographic and military tasks, including oceanic surveillance (e.g., to monitor pollution levels and ecosystems), undersea mapping and mine detection. Limitations due to their operating environment (e.g., impossibility to maintain UUV-operator communication during missions and unexpected changes) require that UUV systems are self-adaptive. These systems are often mission critical (e.g., when used for mine detection) or business critical (e.g., they carry expensive equipment that should not be lost).

The self-adaptive system we use consists of a UUV deployed to carry out a data gathering mission. The UUV is equipped with $n \geq 1$ on-board sensors that can measure the same characteristic of the ocean environment (e.g., water current, salinity or temperature). When used, the sensors take measurements with different, variable rates r_1, r_2, ..., r_n. The probability that each sensor produces measurements that are sufficiently accurate for the purpose of the mission depends on the UUV speed sp, and is given by p_1, p_2, ..., p_n. For each measurement taken, a different amount of energy is consumed, given by e_1, e_2, ..., e_n. Finally, the n sensors can be switched on and off individually (e.g., to save battery power when not required), but these operations consume an amount of energy given by e_1^{on}, e_2^{on}, ..., e_n^{on} and e_1^{off}, e_2^{off}, ..., e_n^{off}, respectively. The UUV must adapt to changes in the sensor measurement rates r_1, r_2, ..., r_n and to sensor failures by dynamically adjusting:

(a) the UUV speed sp

(b) the sensor configuration x_1, x_2, ..., x_n (where $x_i = 1$ if the i-th sensor is on and $x_i = 0$ otherwise)

in order to meet the quality-of-service requirements below:

R1 (throughput): The UUV should take at least 20 measurements of sufficient accuracy for every 10 metres of mission distance.

R2 (resource usage): The energy consumption of the sensors should not exceed 120 Joules per 10 surveyed metres.

R3 (cost): If requirements R1 and R2 are satisfied by multiple configurations, the UUV should use one of these configurations that minimises the cost function

$$cost = w_1 E + w_2 sp^{-1}, \tag{1}$$

where E is the energy used by the sensors to survey a 10m mission distance, and $w_1, w_2 > 0$ are weights that reflect the relative importance of carrying out the mission with reduced battery usage and completing the mission faster.[2]

R4 (safety): If a configuration that meets requirements **R1**–**R3** is not identified within 2 seconds after a sensor rate change, the UUV speed must be reduced to 0m/s. This ensures that the UUV does not advance more than the distance it can cover at its maximum speed within 2 seconds without taking appropriate measurements, and waits until the controller identifies a suitable configuration (e.g., after the UUV sensors recover) or new instructions are provided by a human operator.

3.2 Foreign Exchange Trading System

The service-based system from the area of foreign exchange trading is taken from our recent work in [57]. This system, which we anonymise as FX for confidentiality reasons, is used by an European foreign exchange brokerage company. The FX system implements the workflow shown in Fig. 5 and described below.

An FX customer (called a trader) can use the system in two operation modes. In the *expert* mode, FX executes a loop that analyses market activity, identifies patterns that satisfy the trader's objectives, and automatically carries out trades. Thus, the *Market watch* service extracts real-time exchange rates (bid/ask price) of selected currency pairs. This data is used by a *Technical analysis* service that evaluates the current trading conditions, predicts future price movement, and decides if the trader's objectives are: (i) "satisfied" (causing the invocation of an *Order* service to carry out a trade); (ii) "unsatisfied" (resulting in a new *Market watch* invocation); or (iii) "unsatisfied with high variance" (triggering an *Alarm* service invocation to notify the trader about discrepancies/opportunities not covered by the trading objectives). In the *normal* mode, FX assesses the economic outlook of a country using a *Fundamental analysis* service that collects, analyses and evaluates information such as news reports, economic data and political events, and provides an assessment on the country's currency. If satisfied with this assessment, the trader can use the *Order* service to sell or buy currency, in which case a *Notification* service confirms the completion of the trade. We assume that the FX system has to dynamically select third-party implementations for each service from Fig. 5, in order to meet the following system requirements:

2. Cost (or *utility*) functions that employ weights to combine several performance, reliability, resource use and other quality attributes of software—accounting for differences in attribute value ranges and relative importance—are extensively used in self-adaptive software systems (e.g. [15], [43], [54], [95], [118]).

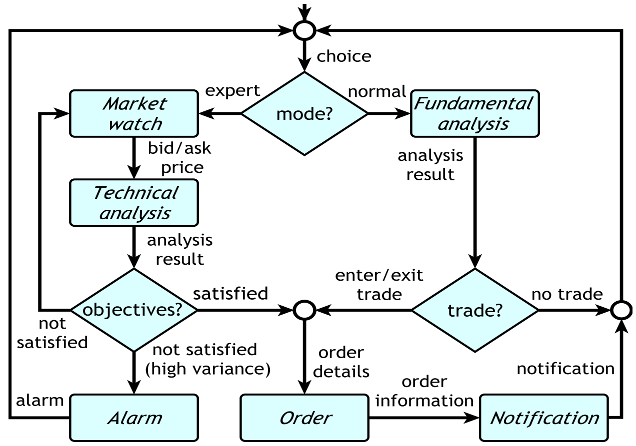

Fig. 5. Foreign exchange trading (FX) workflow

R1 (reliability): Workflow executions must complete successfully with probability at least 0.9.

R2 (response time): The total service response time per workflow execution must be at most 5s.

R3 (cost): If requirements R1 and R2 are satisfied by multiple configurations, the FX system should use one of these configurations that minimises the cost function:

$$cost = w_1 price + w_2 time, \tag{2}$$

where $price$ and $time$ represent the total price of the services invoked by a workflow execution and the response time for a workflow execution, respectively, and $w_1, w_2 > 0$ are weights that encode the desired trade-off between price and response time.

R4 (safety): If a configuration that ensures requirements **R1**–**R3** cannot be identified within 2s after a change in service characteristics is signalled by the sensors of the self-adaptive FX system, the *Order* service invocation is bypassed, so that the FX system does not carry out any trade that might be based on incorrect or stale data.

Note that requirements R1–R3 express two constraints and an optimisation criterion that are qualitatively different from those specified by the requirements from our first case study (cf. Section 3.1). Nevertheless, our tool-supported instance of the ENTRUST methodology enabled the development of the self-adaptive FX system as described in Section 6.

4 The ENTRUST Methodology

The ENTRUST methodology supports the systematic engineering and assurance of self-adaptive systems based on monitor-analyse-plan-execute (MAPE) control loops. This is by far the most common type of control loop used to devise self-adaptive software systems [14], [36], [37], [43], [68], [78], [83], [95]. The engineering of self-adaptive systems based on essentially different control techniques, such as the control theoretical paradigm [97], as for example proposed in [49], is not supported by our methodology.

ENTRUST comprises the tool-supported design-time stages and the automated runtime stages shown in Fig. 6, and is underpinned by two key principles:

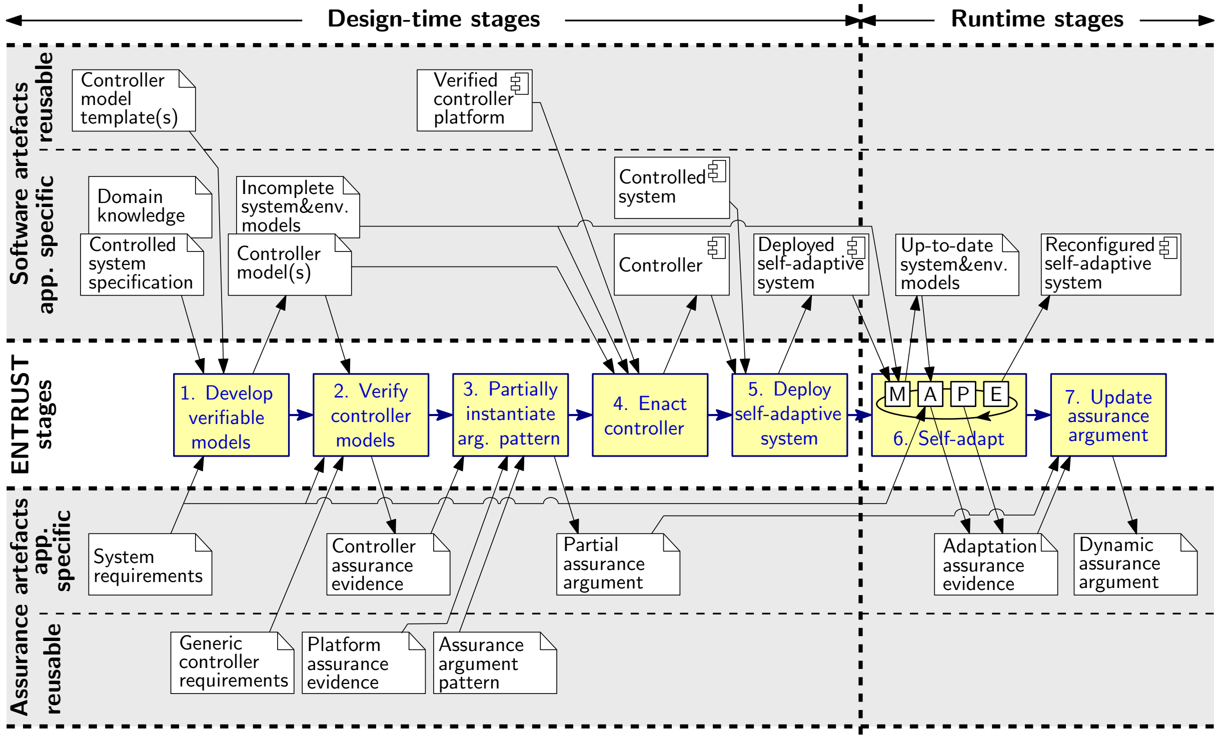

Fig. 6. Stages and key artefacts of the ENTRUST methodology. In line with the two principles underpinning the methodology, its first stage involves the development of verifiable models for the controller, controlled system and environment of the self-adaptive system used throughout the remaining stages, and multiple stages reuse application-independent software and assurance artefacts.

1) *Model-driven engineering is essential for developing trustworthy self-adaptive systems and their assurance cases.* As emphasised in the previous section, model-based analysis, simulation, testing and formal verification—at design time and during reconfiguration—represent the main sources of assurance evidence for self-adaptive software. As such, both the design-time and the runtime stages of our methodology are model driven. Models of the structure and behaviour of the functional components, controller and environment are the basis for the engineering and assurance of ENTRUST self-adaptive systems.
2) *Reuse of application-independent software and assurance artefacts significantly reduces the effort and expertise required to develop trustworthy self-adaptive systems.* Assembling an assurance case for a software system is a costly process that requires considerable effort and expertise. Therefore, the reuse of both software and assurance artefacts is essential for ENTRUST. In particular, the reuse of application-independent controller components and of templates for developing application-specific controller elements also enables the reuse of assurance evidence that these software artefacts are trustworthy.

The ENTRUST stages and their exploitation of these two principles are described in the remainder of this section.

4.1 Design-time ENTRUST Stages

4.1.1 Stage 1: Development of Verifiable Models

In ENTRUST, the engineering of a self-adaptive system with the architecture from Fig. 1 starts with the development of models for:

1) The controller of the self-adaptive system;
2) The relevant aspects of the controlled software system and its environment.

A combination of structural and behavioural models may be produced, depending on the evidence needed to assemble the assurance case for the self-adaptive system under development. ENTRUST is not prescriptive in this respect. However, we require that these models are *verifiable*, i.e., that they can be used in conjunction with methods such as model checking or simulation, to obtain evidence that the controller and the self-adaptive system meet their requirements. As an example, finite state transition models may be produced for the controllers of our UUV and FX systems from Section 3, enabling the use of model checking to verify that these controllers are deadlock free.

The verifiable models are application-specific. As illustrated in Fig. 6, their development requires *domain knowledge*,[3] is based on a *controlled system specification*, and is informed by the *system requirements*. As in other areas of software engineering, we envisage that tool-supported methods will typically be used to obtain these models. However, their manual development or fully automated synthesis are not precluded by ENTRUST.

In line with the "reuse of artefacts" principle, ENTRUST exploits the fact that the controllers of self-adaptive systems implement the established MAPE workflow, and uses application-independent *controller model template(s)* to devise the *controller model(s)*. These templates model the generic aspects of the MAPE workflow and contain placeholders for the application-specific elements of an ENTRUST controller.

3. The ENTRUST software and assurance artefacts that appear in *italics* in the text are also shown in Fig. 6.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSE.2017.2738640

Given the environmental and internal uncertainty that characterises self-adaptive systems, only *incomplete system and environment models* can be produced in this ENTRUST stage. These incomplete models may include unknown or estimated parameters, nondeterminism (i.e., alternative options whose likelihoods are unknown), parts that are missing, or some combination of all of these. For example, parametric Markov chains may be devised to enable the runtime analysis of the requirements for our UVV and FX systems detailed in Sections 3.1 and 3.2, respectively, by means of probabilistic model checking or simulation.

4.1.2 Stage 2: Verification of Controller Models

The main role of the second ENTRUST stage is to produce *controller assurance evidence*, i.e., compelling evidence that a controller based on the controller model(s) from Stage 1 will satisfy a set of *generic controller requirements*. These are requirements that must be satisfied in any self-adaptive system (e.g., deadlock freeness) and are predefined in a format compatible with that of the controller model templates and with the method that will be used to verify the controller models. For example, if labelled transition systems are used to model the controller and model checking to establish its correctness as in [40], [41], these generic controller requirements can be predefined as temporal logic formulae.

The controller assurance evidence must include evidence that the system requirements for application-specific failsafe operating mode(s) are always satisfied. In this way, a minimal assurance case is always available for the scenario when the runtime assurance evidence for other system requirements cannot be obtained and the self-adaptive system needs to switch to a degraded, failsafe mode of operation. Several fallback levels as proposed in [40] can also be supported in this way, with only the most degraded fallback level ensured through assurance evidence obtained in this ENTRUST stage. For example, requirements R4 of our UUV and FX systems from Section 3 specify failsafe operating modes for the two systems, so we will need to show that these requirements are always met.

The assurance evidence generated in this stage of the methodology may be obtained using a range of methods that include formal verification, theorem proving and simulation. The methods that can be used depend on the types of models produced in the previous ENTRUST stage, and on the generic controller requirements and system requirements for which assurance is sought. The availability of tool support in the form of model checkers, theorem provers, SMT solvers, domain-specific simulators, etc. will influence the choice of these methods.

Preparing the design-time models, i.e., developing verifiable models and verifying the controller models, comes with a cost. This cost can be reduced by using tool-supported methods and by exploiting reusable application-independent software, as done by the related approaches described in Section 8. Furthermore, these related approaches that only provide a fraction of the assurances that ENTRUST achieves (as detailed when we discuss related work in Section 8) operate with design-time models that require a comparable effort to specify the models and provide the controller assurance evidence.

4.1.3 Stage 3: Partial Instantiation of Assurance Argument Pattern

This ENTRUST stage uses the controller assurance evidence from Stage 2 to support the partial instantiation of a generic *assurance argument pattern* for self-adaptive software. As explained in Section 2.3, this pattern is an incomplete assurance argument containing placeholders for the system-specific assurance evidence. A subset of the placeholders correspond to the controller assurance evidence obtained in Stage 2, and are therefore instantiated using this evidence. The result is a *partial assurance argument*, which still contains placeholders for the assurance evidence that cannot be obtained until the uncertainties associated with the self-adaptive system are resolved at runtime.

For example, the partial assurance argument for our UUV and FX systems should contain evidence that their controllers are deadlock free and that their failsafe requirements R4 are always satisfied. These requirements can be verified at design time. In contrast, requirements R1–R3 for the two systems cannot be verified until runtime, when the controller acquires information about the measurement rates of the UUV sensors and the third-party services available for the FX operations, respectively. Assurance evidence that requirements R1–R3 are satisfied can only be obtained at runtime.

In addition to the two types of placeholders, the assurance argument pattern used as input for this stage includes assurance evidence that is application independent. In particular, it includes evidence about the correct operation of the *verified controller platform*, i.e. the software that implements application-independent controller functionality used to execute the ENTRUST controllers. This *platform assurance evidence* is reusable across self-adaptive systems.

4.1.4 Stage 4: Enactment of the Controller

This ENTRUST stage assembles the *controller* of the self-adaptive system. The process involves integrating the verified controller platform with the application-specific controller elements, and with the sensors and effectors that interface the controller with the controlled software system from Fig. 1.

The application-specific controller elements must be devised from the verified controller models, by using a trusted model-driven engineering method. This can be done using *model-to-text transformation*, a method that employs a trusted *model compiler* to generate a low-level executable representation of the controller models. Alternatively, the ENTRUST verified controller platform may include a trusted virtual machine[4] able to directly interpret and run the controller models. The second, *model interpretation* method [101], has the advantage that it eliminates the need to generate controller code and to provide additional assurances for it.

4.1.5 Stage 5: Deployment of the Self-Adaptive System

In the last design-time stage, the integrated controller and *controlled components* of the self-adaptive system are installed, preconfigured and activated by means of an

4. Throughout the paper, the term "virtual machine" refers to a software component capable to interpret and execute controller models, much like a Java virtual machine executes Java code.

application-specific process. The pre-configuration is responsible for setting the deployment-specific parameters and architectural aspects of the system. For example, the pre-configuration of the UUV system from Section 3.1 involves selecting the initial speed and active sensor set for the UUV, whereas for the FX system from Section 3.2 it involves choosing initial third-party implementations for each FX service.

The *deployed self-adaptive system* will be fully configured and a complete assurance argument will be available only after the first execution of the MAPE control loop. This execution is typically triggered by the system activation, to ensure that the newly deployed self-adaptive system takes into account the current state of its environment as described next.

4.2 Runtime ENTRUST Stages

4.2.1 Stage 6: Self-adaptation

In this ENTRUST stage, the deployed self-adaptive system is dynamically adjusting its parameters and architecture in line with observed internal and environmental changes. To this end, the controller executes a typical MAPE loop that monitors the system and its environment, using the information obtained in this way to resolve the "unknowns" from the incomplete system and environment models. The resulting *up-to-date system and environment models* enable the MAPE loop to analyse the system compliance with its requirements after changes, and to plan and execute suitable reconfigurations if necessary.

Whenever the MAPE loop produces a *reconfigured self-adaptive system*, its analysis and planning steps generate *adaptation assurance evidence* confirming the correctness of the analysis results and of the reconfiguration plan devised on the basis of these results. This assurance evidence is a by-product of analysis and planning methods that may include runtime verification, simulation and runtime model checking. Irrespective of the methods that produce it, the adaptation assurance evidence is essential for the development of a complete assurance argument in the next ENTRUST stage.

4.2.2 Stage 7: Synthesis of Dynamic Assurance Argument

The final ENTRUST stage uses the adaptation correctness evidence produced by the MAPE loop to fill in the placeholders from the partial assurance argument, and to devise the complete assurance case for the reconfigured self-adaptive system. For example, runtime evidence that requirements R1–R3 of the UUV and FX systems from Section 3 are met will be used to complete the remaining placeholders from their partial assurance arguments. Thus, an ENTRUST assurance case is underpinned by a *dynamic assurance argument* that is updated after each reconfiguration of the system parameters and architecture. This assurance case captures both the full assurance argument and the evidence that justifies the active configuration of the self-adaptive system.

The ENTRUST assurance case versions generated for every system reconfiguration have two key uses. First, they allow decision makers and auditors to understand and assess the present and past versions of the assurance case. Second, they allow human operators to endorse major reconfiguration plans in human-supervised self-adaptive systems. This type of self-adaptive systems is of particular interest in domains where human supervision represents an important risk mitigation factor or may be required by regulations. As an example, UK Civil Aviation Authority regulations [110] permit self-adaptation in certain functions (e.g., power management, flight management and collision avoidance) of unmanned aircraft of no more than 20 kg provided that the aircraft operates within the visual line of sight of a human operator.

5 Tool-Supported Instance of ENTRUST

This section presents an instance of ENTRUST in which the stages described in Section 4 are supported by the modelling and verification tools UPPAAL [7] and PRISM [79]. We start with an overview of this tool-supported ENTRUST instance in Section 5.1, followed by a description of each of its stages in Section 5.2.

5.1 Overview

The ENTRUST methodology can be used with different combinations of modelling, verification and controller enactment methods, which may employ different self-adaptive system architectures and types of assurance evidence. This section presents a tool-supported instance of ENTRUST that uses one such combination of methods. We developed this instance of the methodology with the aim to validate ENTRUST and to ease its adoption.

Our ENTRUST instance supports the engineering of self-adaptive systems with the architecture shown in Fig. 7. The reusable verified controller platform at the core of this architecture comprises:

1) A *Trusted Virtual Machine* that directly interprets and executes models of the four steps from the MAPE control loop[5] (i.e., the ENTRUST controller models).
2) A *Probabilistic Verification Engine* that is used to verify stochastic models of the controlled system and its environment during the analysis step of the MAPE loop.

Using the *Trusted Virtual Machine* for controller model interpretation eliminates the need for a model-to-text transformation of the controller models into executable code, which is a complex, error-prone operation. Not having to devise this transformation and to provide assurance evidence for it are major benefits of our ENTRUST instance. Although we still need assurance evidence for the virtual machine, this was obtained when we developed and verified the virtual machine,[6] and is part of the reusable *platform assurance evidence* for the ENTRUST instance.

The *Probabilistic Verification Engine* consists of the verification libraries of the probabilistic model checker PRISM [79] and is used by the analysis step of the MAPE control loop. As such, our ENTRUST instance works with:

5. Hence the controller models are depicted as software components in Fig. 7.

6. This assurance evidence is in the form of a comprehensive test suite and a report describing its successful execution by the virtual machine, both of which are available on our ENTRUST project website at https://www-users.cs.york.ac.uk/simos/ENTRUST/.

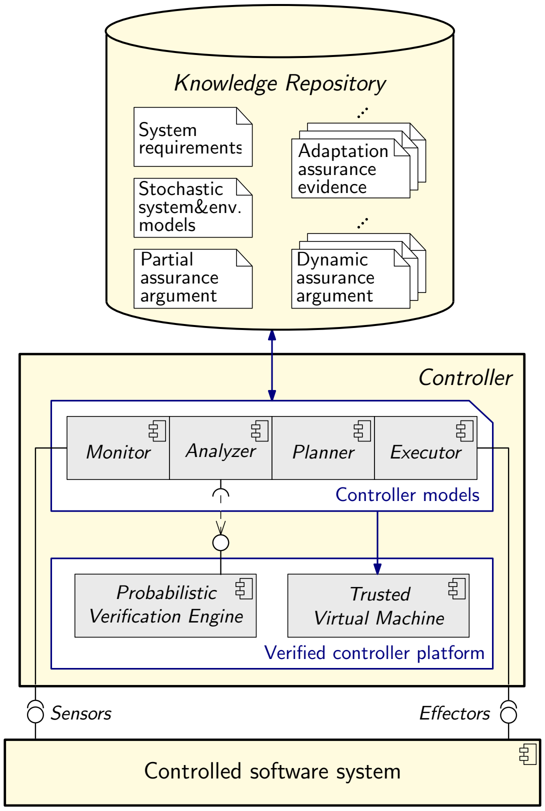

Fig. 7. Architecture of an ENTRUST self-adaptive system

1) Stochastic finite state transition models of the controlled system and the environment, defined in the PRISM high-level modelling language. Incomplete versions of these models are devised in Stage 1 of ENTRUST, and have their unknowns resolved at runtime. All types of models that PRISM can analyse are supported, including discrete- and continuous-time Markov chains (DTMCs and CTMCs), Markov decision processes (MDPs) and probabilistic automata (PAs).
2) Runtime-assured system requirements expressed in the appropriate variant of probabilistic temporal logic, i.e., probabilistic computation tree logic (PCTL) for DTMCs, MDPs and PAs, and continuous stochastic logic (CSL) for CTMCs.

This makes our instantiation of the generic ENTRUST methodology applicable to self-adaptive systems whose non-functional (e.g., reliability, performance, resource usage and cost-related) requirements can be specified in the above logics, and whose behaviour related to these requirements can be described using stochastic models. As shown by the recent work of multiple research groups (e.g., [15], [20], [27], [44], [47], [50], [90], [104]), this represents a broad and important class of self-adaptive software that includes a wide range of service-based systems, web applications, resource management systems, and embedded systems.

Also developed in Stage 1 of ENTRUST, the four controller models form an application-specific network of interacting timed automata [2], and are expressed in the modelling language of the UPPAAL verification tool suite [7].

Accordingly, UPPAAL is used in Stage 2 of ENTRUST to verify the compliance of the controller models with the generic controller requirements and with any system requirements that can be assured at design time. These requirements are defined in computation tree logic (CTL) [31].

In Stage 3 of our ENTRUST instance, a partial assurance argument is devised starting from an assurance argument pattern represented in *goal structuring notation* (GSN) [72].

The controller enactment from Stage 4 involves integrating the timed-automata controller models with our verified controller platform.

In Stage 5 of ENTRUST, the controlled software system and its enacted controller are deployed, together with a *Knowledge Repository* that supports the operation of the controller. Initially, this repository contains: (i) the partial assurance argument from Stage 3; (ii) the system requirements to be assured at runtime; and (iii) the (incomplete) stochastic system and environment models from Stage 1.

During the execution of the MAPE loop in Stage 6 of ENTRUST, the *Monitor* obtains information about the system and its environment through *Sensors*. This information is used to resolve the unknowns from the stochastic models of the controlled system and its environment. Examples of such unknowns include probabilities of transition to 'failure' states for a DTMC, MDP or PA, rates of transition to 'success' states for a CTMC, and sets of states and transitions modelling certain system behaviours. After each update of the stochastic system and environment models, the *Analyzer* reverifies the compliance of the self-adaptive system with its runtime-assured requirements. When the requirements are no longer met, the *Analyzer* uses the verification results to identify a new system configuration that restores this compliance, or to find out that such a configuration does not exist and to select a predefined failsafe configuration. The step-by-step actions needed to achieve the new configuration are then established by the *Planner* and implemented by the *Executor* through the *Effectors* of the controlled system.

Using the *Probabilistic Verification Engine* enables the *Analyzer* and *Planner* to produce assurance evidence justifying their selection of new configurations and of plans for transitioning the system to these configurations, respectively. This adaptation assurance evidence is used to synthesise a fully-fledged, dynamic GSN assurance argument in Stage 7 of our ENTRUST instance. As indicated in Fig. 7, versions of the adaptation assurance evidence and of the dynamic assurance argument justifying each reconfiguration of the self-adaptive system are stored in the *Knowledge Repository*.

The implementation of the ENTRUST stages in our tool-supported instance of the methodology is summarised in Table 2 and described in further detail in Section 5.2.

5.2 Stage Descriptions

5.2.1 Development of Verifiable Models

Controller models. We devised two types of templates for the four controller models from Fig. 7: (i) *event triggered*, in which the monitor automaton is activated by a sensor-generated signal indicating a change in the managed system or the environment; and (ii) *time triggered*, in which the monitor is activated periodically by an internal clock. The event-triggered automaton templates are shown in Fig. 8 using the following font and text style conventions:

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSE.2017.2738640

TABLE 2
Stages of the tool-supported instance of the ENTRUST methodology

Stage	Type	Description	Supporting tool(s)
1	tool supported	Timed automata controller models developed from UPPAAL templates	UPPAAL
		Incomplete stochastic models of the controlled system and environment developed based on system specification and domain knowledge	PRISM
2	tool supported	Controller models verified to obtain controller assurance evidence	UPPAAL
3	manual	Partial assurance argument devised from GSN assurance argument pattern	–
4	manual	Controller enacted by integrating the verified controller models and platform	–
5	manual	Controlled system, controller and knowledge repository deployed	–
6	automated	MAPE control loop continually executed to ensure the system requirements	PRISM & ENTRUST controller platform
7	automated	GSN dynamic assurance argument generated	ENTRUST controller platform

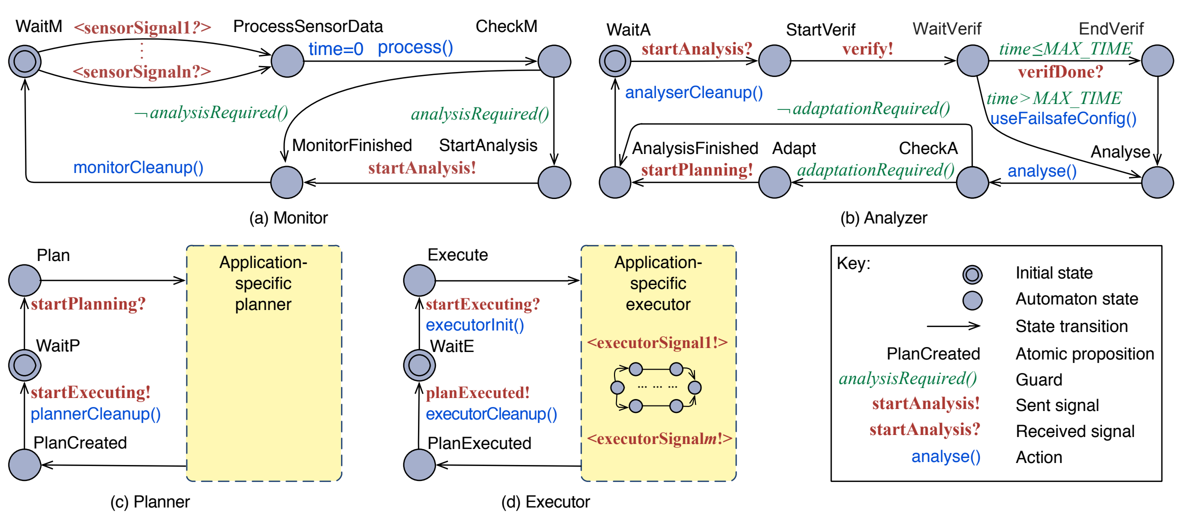

Fig. 8. Event-triggered MAPE model templates

- Sans-serif font is used to annotate states with the atomic propositions (i.e. boolean properties) that hold in those states, e.g. PlanCreated from the Planner automaton;
- *Italics text* is used for the guards that annotate state transitions with the conditions which must hold for the transitions to occur, e.g. *time≤MAX_TIME* from the Analyzer automaton;
- State transitions are additionally annotated with the actions executed upon taking the transitions, and these actions are also shown in sans-serif font, e.g. time=0 to initialise a timer in the Monitor automaton;
- **Bold text** is used for the synchronisation channels between two automata—these channels are specified as pairs comprising a '!'-decorated sent signal and a '?'-decorated received signal with the same name, e.g., **startAnalysis!** and **startAnalysis?** from the monitor and analyzer automata, respectively. The two transitions associated with a synchronisation channel can only be taken at the same time.

Finally, signals in angle brackets '⟨⟩' are placeholders for application-specific signal names, and guards and actions decorated with brackets '()' represent application-specific C-style functions.

To specialise these model templates for a particular system and application, software engineers need: (a) to replace the signal placeholders with real signal names; (b) to define the guard and action functions; and (c) to devise the automaton regions shaded in Fig. 8. For example, for the monitor automaton the engineers first need to replace the placeholders $\langle \textbf{sensorSignal}_1\textbf{?} \rangle, \ldots, \langle \textbf{sensorSignal}_n\textbf{?} \rangle$ with sensor signals announcing relevant changes in the managed system. They must then implement the functions process(), *analysisRequired()* and monitorCleanup(), whose roles are to process the sensor data, to decide if the change specified by this data requires the "invocation" of the analyzer through the **startAnalysis!** signal, and to carry out any cleanup that may be required, respectively. Details about the other automata from Fig. 8 are available on our project website, which also provides implementations of these MAPE model templates in the modelling language of the UPPAAL verification tool suite [7].

Parametric stochastic models. These models used by the ENTRUST control loop at runtime are application specific, and need to be developed from scratch. Their parameters correspond to probabilities or rates of transition between model states, and are continually estimated at runtime, based on change information provided by the sensors of the controlled system. As such, the verification of these models at runtime enables the ENTRUST analyzer to identify

TABLE 3
Stochastic models supported by the ENTRUST instance, with citations of representative research that uses them in self-adaptive systems

Type of stochastic model	Non-functional requirement specification logic
Discrete-time Markov chains [15], [23], [44], [46], [47], [59]	PCTL[a], LTL[b], PCTL*[c]
Markov decision processes [50]	PCTL[a], LTL[b], PCTL*[c]
Probabilistic automata [21], [70]	PCTL[a], LTL[b], PCTL*[c]
Continuous-time Markov chains [19], [22], [55]	CSL[d]
Stochastic games [26], [27]	rPATL[e]

[a] Probabilistic Computation Tree Logic [9], [61]
[b] Linear Temporal Logic [88]
[c] PCTL* is a superset of PCTL and LTL
[d] Continuous Stochastic Logic [3], [4]
[e] reward-extended Probabilistic Alternating-time Temporal Logic [28]

configurations it can use to meet the system requirements after unexpected changes, as described in detail in [15], [20], [22], [44], [46]. The types of stochastic models supported by our ENTRUST instance are shown in Table 3. As illustrated by the research work cited in the table, the temporal logics used to express the properties of these models support the specification of numerous performance, reliability, safety, resource usage and other non-functional requirements that recent surveys propose for self-adaptive systems [30], [117].

To ensure the accuracy of the stochastic models described above, ENTRUST can rely on recent advances in devising these models from logs [59], [87] and UML activity diagrams [17], [52], and in dynamically and accurately updating their parameters based on sensor-provided runtime observations of the controlled system [16], [23], [44], [48].

5.2.2 Verification of Controller Models

During this ENTRUST stage, a trusted model checker is used to verify the network of MAPE automata devised in the previous section. This verification yields evidence that the MAPE models satisfies a set of key safety and liveness properties that include both generic and application-specific properties. Table 4 shows a non-exhaustive list of generic properties that we assembled for the current version of ENTRUST. Although these properties are application-independent, verifying that an ENTRUST controller satisfies them is possible only after its application-specific MAPE models were devised. This involves completing the application-specific parts of the planner and executor automata, and implementing the functions for the *guards* and actions from all the model templates.

Additionally, automata that simulate the controller sensors, runtime probabilistic verification engine and effectors from Fig. 7 need to be defined to enable this verification. The sensors, verification engine and effectors automata have to synchronise with the relevant monitor, analyzer and executor signals, respectively. The sensors automaton and verification automaton also have to exercise the possible paths through the monitor, analyzer and planner automata (and indirectly the executor automaton). To this end, they can nondeterministically populate the knowledge repository with data that satisfies all the different guard combinations. Alternatively, a finite collection of the two automata can be used to verify subsets of all possible MAPE paths, as long as the union of all such subsets covers the entire behaviour space of the MAPE network of automata.

Note that these application-specific elements of the MAPE automata are much larger than the application-independent elements from the MAPE model templates. Therefore, we do not use compositional model checking [32], [70] to verify the two parts of the MAPE automata separately, with the application-independent elements verified once and for all. Such an approach would increase the complexity of the verification task (e.g. by requiring the identification and verification of less intuitive "assumptions" [33] that the application-specific parts of the automata need to "guarantee") without any noticeable reduction in the verification time, almost all of which would be required to verify the application-specific automata elements.

5.2.3 Partial Instantiation of Assurance Argument Pattern

We used the *Goal Structuring Notation* (GSN) introduced in Section 2.2 to devise a reusable *assurance argument pattern* (cf. Section 2.3) for self-adaptive software. Unlike all existing assurance argument patterns [63], our new pattern captures the fact that for self-adaptive software the assurance process cannot be completed at design time. Instead, it is a continual process where some design features and code elements are dynamically reconfigured and executed during self-adaptation. As such, the detailed claims and evidence for meeting the system requirements must vary with self-adaption, and thus ENTRUST assurance cases must evolve dynamically at runtime.

The ENTRUST assurance argument pattern is shown in Fig. 9. Its root goal, **ReqsSatisfied**, states that the system requirements are satisfied at all times. These requirements are typically allocated to the software from the higher-level system analysis process, so the justifications of their derivation, validity and completeness are addressed as part of the overall system assurance case (which is outside the scope of the software assurance case). **ReqsSatisfied** is supported by a sub-claim based on (i.e. in the context of) the current configuration (**ReqsConfiguration**) and by a reconfiguration sub-claim (**Reconfig**). That is, the pattern shows that we are guaranteeing that the current configuration satisfies the requirements (in the absence of changes) and that the ENTRUST controller will plan and execute a reconfiguration that will satisfy these requirements (should a change occur).

The pattern justifies how the system requirements are achieved for each configuration by using a sub-goal **Rx-Achieved** for each requirement Rx. Further, a new configuration has the potential to introduce erroneous behaviours (e.g., deadlocks). The justification for the absence of these errors is provided via the away goal **NoErroneousBehaviour** (described below). The pattern concludes with the goals **RxVerified** and **ReqsPreservedByPlatform**, which justify the verification and the implementation of the formalised requirements, respectively. The away goal **ReqsPreserved-ByPlatform** confirms that the controlled system handles correctly the reconfiguration commands received through effectors. This away goal is obtained using standard assurance processes, which are outside the scope of this paper.

As shown Fig. 10, the **NoErroneousBehaviour** away goal is supported by two sub-claims. The **FMsManaged**

TABLE 4
Generic properties that should be satisfied by an ENTRUST controller

ID	Informal description	Specification in computation tree logic (CTL) [31]
P1	The ENTRUST controller is deadlock free.	A□ not deadlock
P2	Whenever analysis is required, the Analyser eventually carries out this action.	A□ (Monitor.StartAnalysis → A◊ Analyzer.Analyse)
P3	Whenever the system requirements are violated, a stepwise reconfiguration plan is eventually assembled.	A□ (Analyzer.Adapt → A◊ Planner.PlanCreated)
P4	Whenever a stepwise plan is assembled, the Executor eventually implements it.	A□ (Planner.PlanCreated → A◊ Executor.PlanExecuted)
P5	Whenever the Monitor starts processing the received data, it eventually terminates its execution.	A□ (Monitor.ProcessSensorData → A◊ Monitor.Finished)
P6	Whenever the Analyser begins the analysis, it eventually terminates its execution.	A□ (Analyzer.Analyse → A◊ Analyzer.AnalaysisFinished)
P7	A plan is eventually created, each time the Planner starts planning.	A□ (Planner.Plan → A◊ Planner.PlanCreated)
P8	Whenever the Executor starts executing a plan, the plan is eventually executed.	A□ (Executor.Execute → A◊ Executor.PlanExecuted)
P9	Whenever adaptation is required, the current configuration and the best configuration differ.	A□ (Analyzer.Adapt → currentConfig != newConfig)

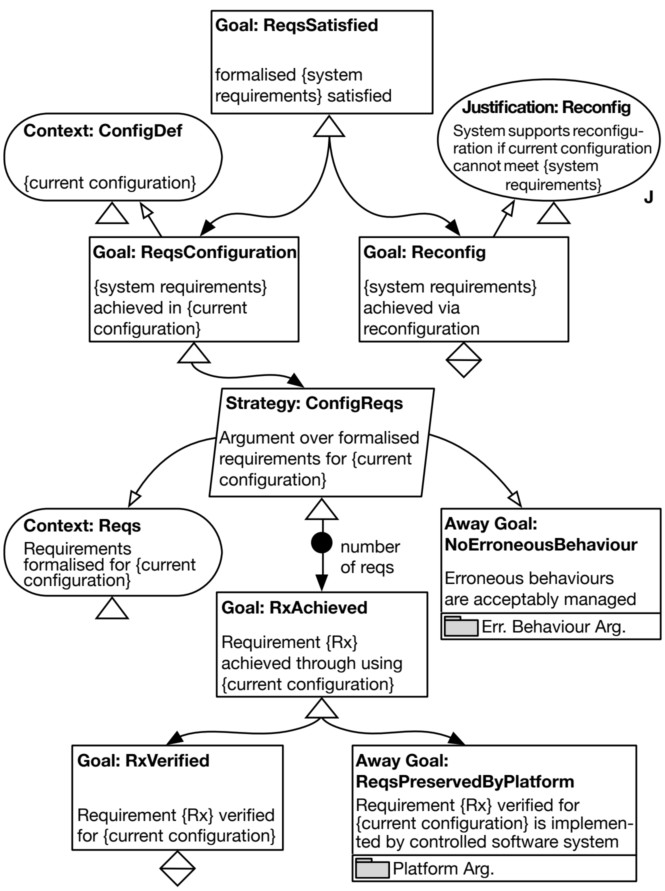

Fig. 9. ENTRUST assurance argument pattern.

sub-claim uses the goals **FMsIdentified** and **ReqsDerived** to state that the relevant "failure modes" for the self-adaptive system have been identified and that the system requirements fully address these failure modes. We leave the two goals undeveloped, as they are achieved using standard requirements engineering and assurance practices. The **EngErrorsAbsent** sub-claim states that the engineering of the self-adaptive system does not introduce errors in the context of the ENTRUST reusable artefacts (i.e., of our trusted virtual machine and probabilistic verification engine) and of the generic properties that an ENTRUST controller has to satisfy. **EngErrorsAbsent** is in turn supported by two sub-goals, **NoProcessError** and **NoController&SystemError**. The former sub-goal is obtained through using suitable software engineering processes (via the away goal **SuitableSoftEngProcess**, which also covers the use of the methods mentioned in Section 5.2.1 to ensure the accuracy of the ENTRUST stochastic models) and through avoiding methodological errors by using the ENTRUST methodology. The latter sub-goal, **NoController&SystemError**, is achieved by claims about:

1) The absence of controller errors. This is supported by (i) the controller verification evidence from Stage 2 of ENTRUST; and (ii) the reusable platform assurance evidence, which includes (testing) evidence about the correct operation of the model checkers UPPAAL and PRISM, based on their long track record of successful adoption across multiple domains and on our own experience of using them to develop self-adaptive systems.
2) The absence of controlled system errors, covered by the **ControlledSystem** away goal.

The away goals **SuitableSoftEngProcess** and **ControlledSystem** are obtained following existing software assurances processes, and thus we do not describe them here.

The partial instantiation of the assurance argument pattern in the last design-time stage of ENTRUST produces a *partially-developed and partially-instantiated* assurance argument [38]. This includes placeholders for items of evidence that can only be instantiated and developed based on operational data, i.e., the runtime verification evidence that is generated by the analysis and planning steps of the ENTRUST controller.

5.2.4 Enactment of the Controller

In this stage, the controller from Fig. 7 is assembled by integrating the MAPE controller models discussed in Sec-

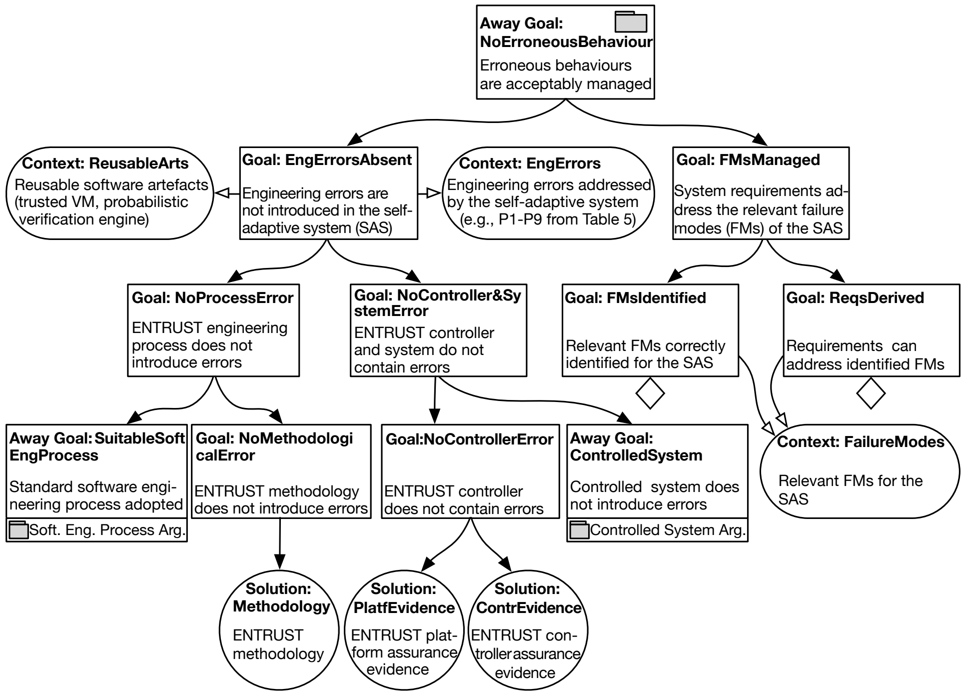

Fig. 10. Away goal **NoErroneousBehaviour**, which justifies the absence of errors due to reconfiguration and is based on the existing GSN pattern Hazardous Contribution Software Safety Argument from the existing GSN catalogue [63]

tion 5.2.1, the ENTRUST verified controller platform and application-specific sensor, effector and stochastic model management components. The application-specific components include generic functionality such as the signals through which these components synchronise with the MAPE automata (e.g., **verify?** and **planExecuted?**). Accordingly, our current version of ENTRUST includes abstract Java classes that provide this common functionality. These abstract classes, which we made available on the project website, need to be specialised for each application. Thus, the specialised sensors and effectors must use the APIs of the managed software system to observe its state and environment, and to modify its configuration, respectively. The stochastic model management component must specialise the probabilistic verification engine so that it instantiates the parametric stochastic models using the actual values of the managed system and environment parameters (provided by sensors) and analyses the application-specific requirements.

5.2.5 Deployment of the Self-Adaptive System

As explained in Section 4.1.5, the role of this stage is to integrate the ENTRUST controller and the controlled software system into a self-adaptive software system that is then installed, preconfigured and set running. In particular, the pre-configuration must select initial values for all the parameters of the controlled system. Immediately after it starts running and until the first execution of the MAPE control loop, the system functions as a traditional, non-adaptive software system. As such, a separate assurance argument (which is outside the scope of this paper) must be developed using traditional assurance methods, to confirm that the initial system configuration is suitable.

The newly running software starts to behave like a self-adaptive system with the first execution of the MAPE control loop, as described in the next two sections.

5.2.6 Self-Adaptation

In this ENTRUST stage, the deployed self-adaptive system is dynamically adjusting its configuration in line with the observed internal and environmental changes. The use of continual verification within the ENTRUST control loop produces assurance evidence that underpins the dynamic generation of assurance cases in the next stage of our ENTRUST instance.

5.2.7 Synthesis of Dynamic Assurance Argument

The ENTRUST assurance case evolves in response to the results of the MAPE process, e.g., *time-triggered* and *event-triggered* outputs of the monitor, the outcomes of the analyzer, the mitigation actions developed by the planner and their realisation by the executor. This offers a dynamic approach to assurance because the full instantiation of the ENTRUST assurance argument pattern is left to runtime, i.e. the only stage when the evidence required to complete the argument becomes available. As such, the assurance case resulting from this stage captures the full argument and evidence for the justification of the current configuration of the self-adaptive system.

6 Applying the ENTRUST Methodology

6.1 Development of Verifiable Models

6.1.1 UUV System

Controller models. We instantiated the ENTRUST model templates for the UUV system from Section 3.1, obtaining the automata shown in Fig. 11. The signal **newRate?** is the only sensor signal that the monitor automaton needs to deal with, by reading a new UUV-sensor measurement rate (in process()) and checking whether this rate has changed to such extent that a new analysis is required

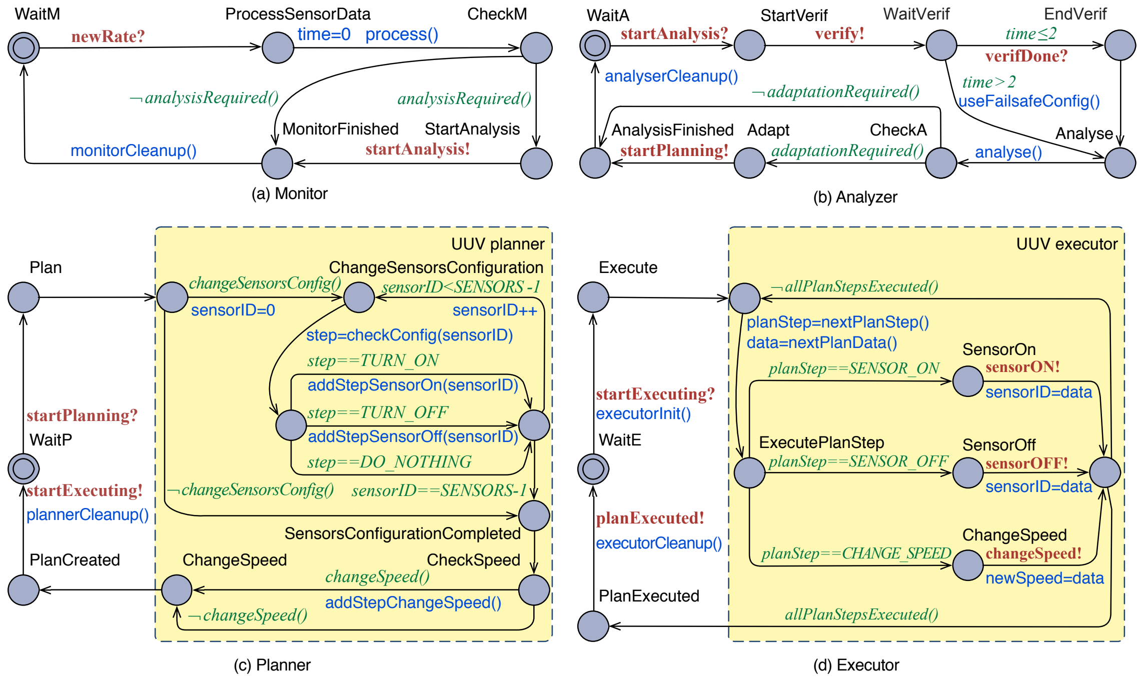

Fig. 11. UUV MAPE automata that instantiate the event-triggered ENTRUST model templates

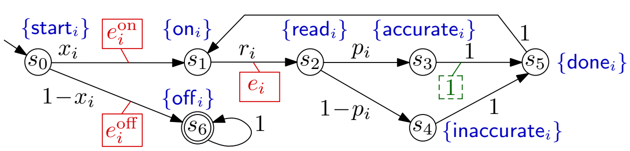

Fig. 12. CTMC model M_i of the i-th UUV sensor, adopted from [55]

(in *analysisRequired()*). If analysis is required, the analyzer automaton sends a **verify!** signal to invoke the runtime verification engine, and thus verifies which UUV configurations satisfy requirements R1 and R2 and with what *cost*. The function analyse() uses the verification results to select a configuration that satisfies R1 and R2 with minimum *cost* (cf. requirement R3). If no such configuration exists or the verification does not complete within 2 seconds and the guard '*time*>2' is triggered, a zero-speed configuration is selected (cf. requirement R4). If the selected configuration is not the one in use, *adaptationRequired()* returns true and the **startPlanning!** signal is sent to initiate the execution of the planner automaton. The planner assembles a stepwise plan for changing to the new configuration by first switching on any UUV sensors that require activation, then switching off those that are no longer needed, and finally adjusting the UUV speed. These reconfiguration steps are carried out by the executor automaton by means of **sensorON!**, **sensorOFF!** and **changeSpeed!** signals handled by the effectors from Fig. 7, as described in Section 5.2.4.

Parametric stochastic models. Fig. 12 shows the CTMC model M_i of the i-th UUV sensor. From the initial state s_0, the system transitions to state s_1 or s_6 if the sensor is switched on ($x_i = 1$) or off ($x_i = 0$), respectively. The sensor takes measurements with rate r_i, as indicated by the transition $s_1 \to s_2$. A measurement is accurate with probability p_i as shown by the transition $s_2 \to s_3$; when inaccurate, the transition $s_2 \to s_4$ is taken. While the sensor is active this operation is repeated, as modelled by the transition $s_5 \to s_1$. The model is augmented with two *reward structures*. A "measure" structure, shown in a dashed rectangular box, associates a reward of 1 to each accurate measurement taken. An "energy" structure, shown in solid rectangular boxes, associates the energy used to switch the sensor on (e_i^{on}) and off (e_i^{off}) and to perform a measurement (e_i) with the transitions modelling these events. The model M of the n-sensor UUV is given by the parallel composition of the n sensor models: $M = M_1||...||M_n$; and the QoS system requirements are specified using CSL as follows:

R1: $R^{\text{measure}}_{\geq 20}[C^{\leq 10/sp}]$

R2: $R^{\text{energy}}_{\leq 120}[C^{\leq 10/sp}]$

R3: minimise$(w_1E+w_2sp^{-1})$, where $E = R^{\text{energy}}_{=?}[C^{\leq 10/sp}]$

where $10/sp$ is the time taken to travel 10m at speed sp. As requirement R4 is a failsafe requirement, we verify it at design time as explained in Section 6.2.1, so it is not encoded into CSL.

6.1.2 FX System

Controller models. We specialised our event-triggered MAPE model templates for the FX system. The resulting MAPE models are shown in Fig. 13, where the shaded areas in Planner and Executor automata indicate the FX-specific steps for assembling a plan and executing the adaptation, respectively. The implementations of all *guards* and actions decorated with brackets '()' (which represent application-specific C-style functions, as explained in Section 5.2.1) are available on our project website.

Parametric stochastic models. To model the runtime behaviour of the FX system, we used the parametric discrete-time Markov chain (DTMC) depicted in Fig. 14. In this DTMC, constant transition probabilities derived from system logs are associated with the branches of the FX work-

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSE.2017.2738640

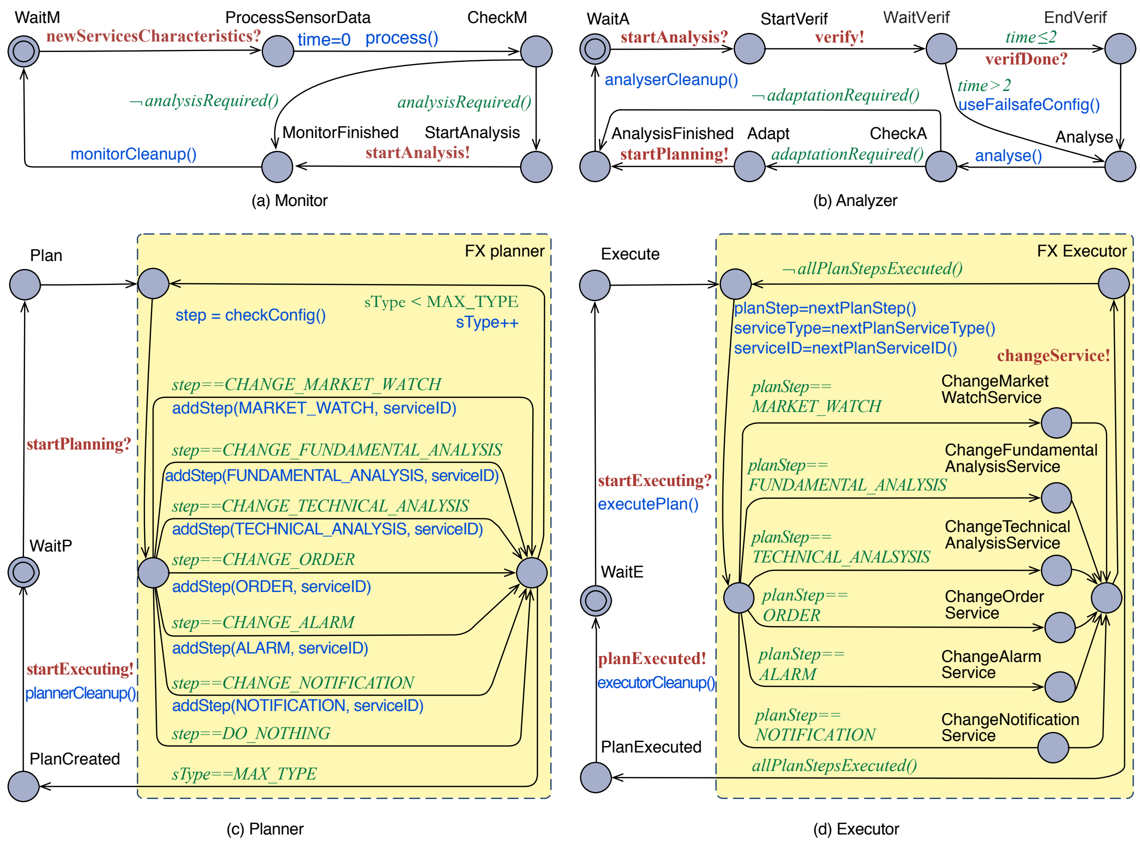

Fig. 13. FX MAPE automata that instantiate the event-triggered ENTRUST model templates

flow from Fig. 5. In contrast, state transitions that model the success or failure of service invocations are associated with parameterised probabilities, which are unknown until the runtime selection of the FX services. Likewise, the "price" and (response) "time" reward structures (shown in solid and dashed boxes, respectively) are parametric and depend on the combination of FX services dynamically selected by the ENTRUST controller.

Finally, we formalised requirements R1–R3 in rewards-augmented probabilistic computational tree logic (PCTL):

R1: $P_{\geq 0.9}[F\,\text{done}]$

R2: $R^{\text{time}}_{\leq 5}[F\,\text{done}]$

R3: minimise $(w_1 price + w_2 time)$, where
$price = R^{\text{price}}_{=?}[F\,\text{done}]$ and $time = R^{\text{time}}_{=?}[F\,\text{done}]$

6.1.3 *Discussion*

The ENTRUST controller model templates supported the development of the UUV and FX controller models with structural changes confined to the Planner and Executor automata. Despite the differences between the sensor data used by the two systems (cf. Table 1), the Monitor and Analyzer automata could be instantiated with all application-specific functionality provided by the guard and action functions associated with the automata transitions. Different types of stochastic models were required for the two systems (continuous time for the UUV system, and discrete time for the FX system) as the differences in their requirements and uncertainties needed the modelling of different aspects of their behaviour.

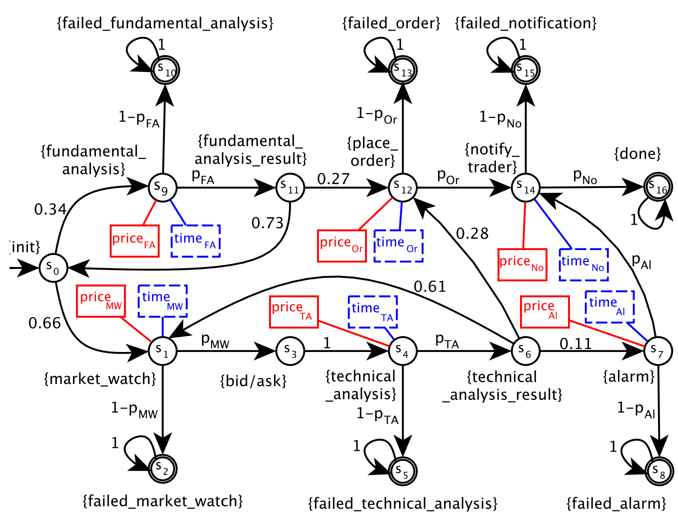

Fig. 14. Parametric DTMC model of the FX system; p_{MW}, p_{TA}, ..., $time_{\text{MW}}$, $time_{\text{TA}}$, ..., and $price_{\text{MW}}$, $price_{\text{TA}}$, ..., represent the *reliability* (i.e. success probability), the *response time* and the *price*, respectively, of the implementations used for the MW, TA, ... system services.

6.2 Verification of Controller Models

6.2.1 *UUV System*

We used the UPPAAL model checker [7] to verify that the network of MAPE automata from Fig. 11 (which we made available on our project website) satisfies all the generic correctness properties from Table 4, as well as the application-specific property

R4: A□ (Analyzer.Analyse ∧ Analyzer.time>2 →
A◊ Planner.Plan ∧ newConfig.speed==0),

which represents the CTL encoding of requirement R4. To carry out this verification, we defined simple sensors, verification engine and effectors automata as described above. We used a simple one-state effectors automaton with transitions returning to its single state for each of the received signals **sensorON?**, **sensorOFF? changeSpeed?** and **planExecuted?**; and a finite collection of sensor–verification engine automata pairs that together exercised all possible paths of the MAPE automata from Fig. 11. These auxiliary UPPAAL automata are available on the project website.

6.2.2 FX System

We used the model checker UPPAAL to verify that the MAPE automata network from Fig. 13 satisfies the generic controller correctness properties in Table 4, and a FX-specific CSL property corresponding to the failsafe requirement R4 of the FX system:

R4: A□ (Analyzer.Analyse ∧ Analyzer.time>2 →
A◊ Planner.Plan ∧ newConfig.Order==NoSvc),

where 'newConfig.Order==NoSvc' signifies that no service is used to implement the *Order* operation (i.e., the operation is skipped).

6.2.3 Discussion

The availability of a set of generic properties that must be satisfied by all ENTRUST controllers (cf. Table 4) meant that an additional CSL property was only needed for the application-specific failsafe requirement. For both systems, this additional property corresponds to the scenario where a suitable new configuration cannot be obtained timely, suggesting that using a property template may be feasible for this and potentially for other types of failsafe requirements.

6.3 Partial Instantiation of Assurance Argument Pattern

6.3.1 UUV System

Fig. 15 shows the partially-instantiated assurance argument pattern for the self-adaptive UUV system, in which we shaded the (partially) instantiated GSN elements. To keep the diagram clear, we only show the expansion for requirements R1 and R4, leaving R2 and R3 undeveloped. The goal **R1Achieved** (which needs to be further instantiated when the system configuration is dynamically selected) is supported by: (a) sub-claim **R1Verified**, whose associated solution placeholder **R1Result** remains uninstantiated and should constantly be updated by the ENTRUST controller at runtime; and (b) the away goal **ReqsPreservedByPlatform** described earlier in this section. The undeveloped and partially instantiated goals **R2Achieved** and **R3Achieved** have the same structure as **R1Achieved**. In contrast, the (failsafe) goal **R4Achieved** is fully instantiated because the solution **R4Result**, comprising UPPAAL verification evidence that R4 is achieved irrespective of the configuration of the self-adaptive system, was obtained in the second ENTRUST stage (verification of controller models), cf. Section 6.2.1.

6.3.2 FX System

We partially instantiated the ENTRUST assurance argument pattern for our self-adaptive FX system, as shown in Fig. 16.

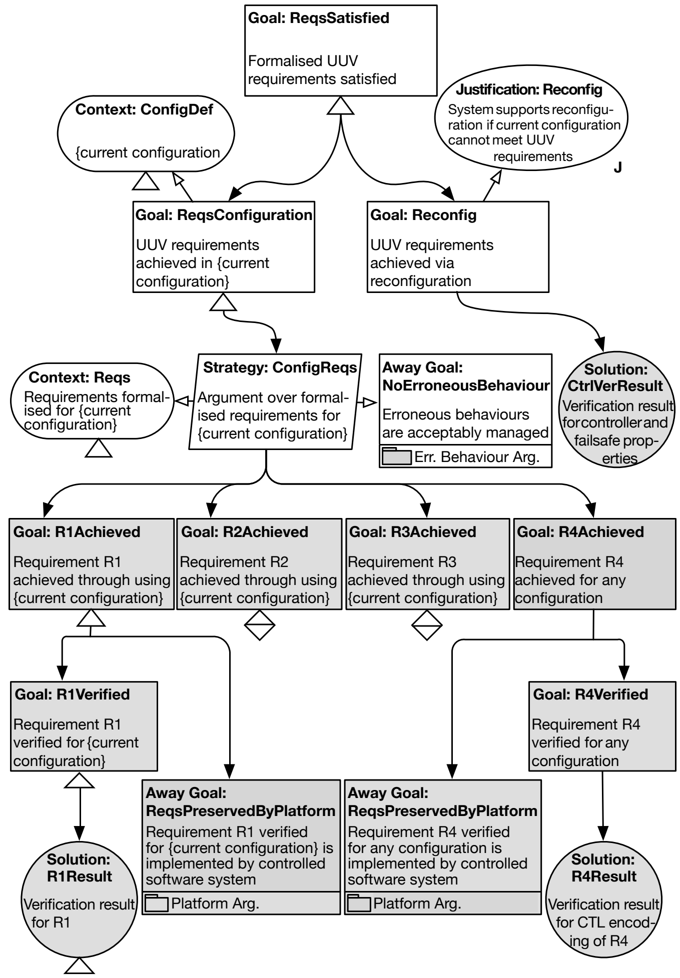

Fig. 15. Partially-instantiated assurance argument for the UUV system

6.3.3 Discussion

As shown in Figs. 15 and 16, roughly the top half of the partially instantiated assurance argument pattern comes from ENTRUST assurance pattern in Fig. 9. This part of the assurance argument captures assurance elements generic to all self-adaptive systems, allowing the developers of a self-adaptive system to focus on the application-specific elements, which they are often more familiar with.

6.4 Enactment of the Controller

6.4.1 UUV System

To assemble an ENTRUST controller for the UUV system, we implemented Java classes that extend the functionality of the abstract Sensors, Effectors and VerificationEngine classes from the ENTRUST distribution. In addition to synchronising with the relevant application-specific signals from the MAPE automata (e.g., **newRate?**), the specialised sensors and effectors invoke the relevant API methods of our UUV simulator. The specialised verification engine instantiates the parametric sensor models M_i from Fig. 12, $1 \leq i \leq n$, and verifies the CSL-encoded requirements from Section 6.1.1.

6.4.2 FX System

To assemble the ENTRUST controller for the FX system, we combined the controller and stochastic models from

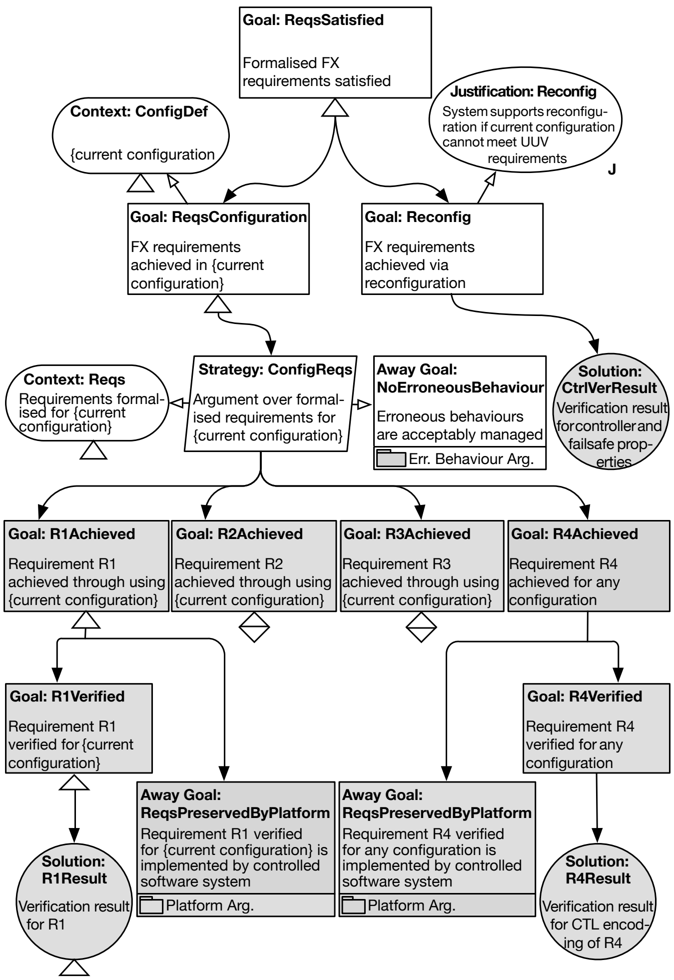

Fig. 16. Partially-instantiated assurance argument for the FX system; the elements (partially) instantiated in Stage 3 of ENTRUST are shaded.

Stage 1 with our generic controller platform, and with FX-specific Java classes that we implemented to specialise the abstract Sensors, Effectors and VerificationEngine abstract classes of ENTRUST. The Sensors class synchronises with the Monitor automaton from Fig. 13 through the **newServicesCharacteristics!** signal (issued after changes in the properties of the FX services are detected). In addition, the Sensors and Effectors classes use the relevant API methods of an FX implementation that we developed as explained in Section 6.5.2. The specialised VerificationEngine instantiates the parametric DTMC model from Fig. 14 at runtime, and verifies the PCTL formulae devised for requirements R1–R3 from Section 6.1.2.

6.4.3 *Discussion*

The controller enactment comprises typical software development (i.e. specialisation of Java classes) and integration tasks. A considerable part of the required functionality is application-independent, and already provided by the reusable abstract Java classes available with ENTRUST.

6.5 Deployment of the Self-Adaptive System

6.5.1 *UUV System*

We used the open-source MOOS-IvP[7] platform (oceanai.mit.edu/moos-ivp) for the implementation of autonomous applications on unmanned marine vehicles [8] to develop a fully-fledged three-sensor UUV simulator that is available on the ENTRUST website. We then exploited the publish-subscribe architecture of MOOS-IvP to interface the ENTRUST sensors and effectors (and thus the controller from Section 6.4.1) with the UUV simulator, we installed the controller and the controlled system on a computer with a similar spec to that of the payload computer of a mid-range UUV, and we preconfigured the system to start with zero speed and all its sensors switched off. We chose this configuration, corresponding to initial UUV parameter values $(x_1, x_2, x_3, sp) = (0, 0, 0, 0)$, to ensure that the system started with a configuration satisfying its failsafe requirement R4 (cf. Section 3.1).[8]

6.5.2 *FX System*

We implemented a prototype version of the FX system using Java web services deployed in Tomcat/Axis, and a Java FX workflow that we integrated with the ENTRUST controller from Stage 4. Our self-adaptive FX system (whose code is available on our project website) could select from two functionally equivalent web service implementations for each of the six FX services from Fig. 5, i.e. from 12 web services with the initial characteristics shown in Table 5. For simplicity and without loss of generality, we installed the components of the self-adaptive FX system on a single computer with the characteristics detailed in Section 7.1, and we preconfigured the system to start by using the first web service implementation available for each service (i.e. MW_0, TA_0, etc.), except for the *Order* service. For *Order*, NoSvc was selected initially, to ensure that the failsafe requirement R4 was satisfied until a configuration meeting requirements R1–R3 was automatically selected by the first execution of the MAPE loop, shortly after the system started.

6.5.3 *Discussion*

This stage involved a typical deployment of the managed systems and of their controllers, except that both self-adaptive systems were preconfigured to start with a configuration satisfying their failsafe requirement. Note that such a configuration always exists because the compliance of the two systems with their failsafe requirements was formally verified in the second ENTRUST stage (cf. Section 6.2).

6.6 Self-Adaptation

6.6.1 *UUV System*

The dynamic reconfiguration of the self-adaptive UUV system is described in detail in Section 7.1.1. Here we illustrate the process by considering a scenario in which the UUV system comprises $n = 3$ sensors with: initial measurement

7. Mission-Oriented Operating Suite – Interval Programming
8. The use of a failsafe initial configuration is our recommended approach for ENTRUST self-adaptive systems. When this is not possible, an execution of the MAPE loop must be initiated as part of the system start-up, to ensure that an initial configuration meeting the system requirements is selected.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSE.2017.2738640

TABLE 5
Initial characteristics of the service instances used by the FX system

Operation:	*Market Watch*		*Technical Analysis*		*Fundam. Analysis*		*Alarm*		*Order*		*Notification*	
Service ID:	MW_0	MW_1	TA_0	TA_1	FA_0	FA_1	Al_0	Al_1	Or_0	Or_1	No_0	No_1
response time [s]	.5	.5	.6	1.0	1.6	.7	.6	.9	.6	1.3	1.8	.5
reliability	.976	.995	.998	.985	.998	.99	.995	.99	.995	.95	.99	.99
price	5	10	6	4	23	25	15	9	25	20	5	8

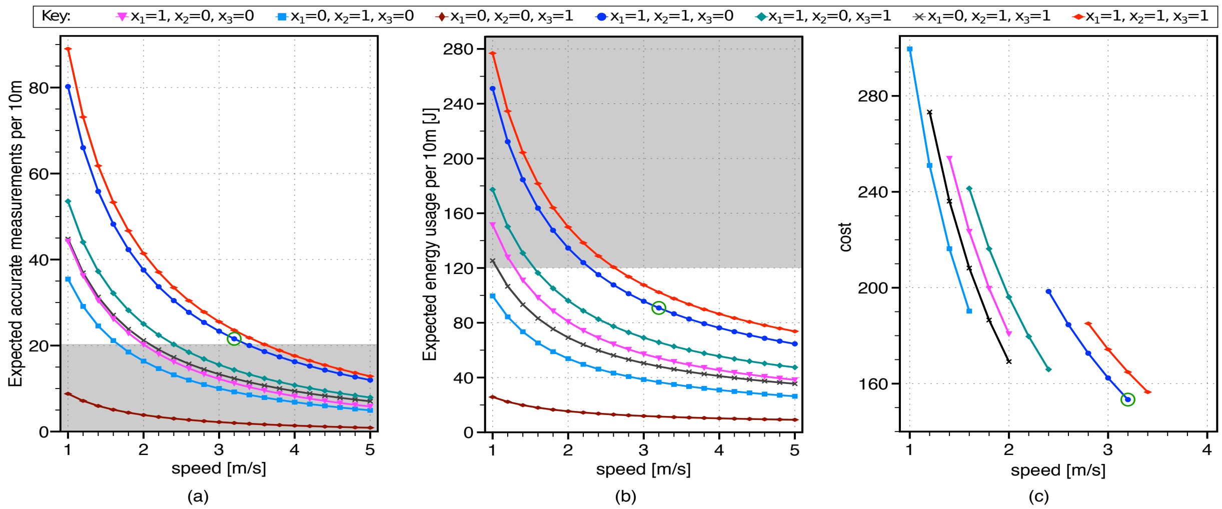

Fig. 17. Verification results for requirement (a) R1, (b) R2, and (c) cost of the feasible configurations; 21 speed values between 1m/s and 5m/s are considered for each of the seven combinations of active sensors, corresponding to $21 \times 7 = 147$ alternative configurations. The best configuration (circled) corresponds to $x_1 = x_2 = 1$, $x_3 = 0$ (i.e. UUV using only its first two sensors) and $sp = 3.2$m/s, and the shaded regions correspond to requirement violations.

rates $r_1 = 5\text{s}^{-1}, r_2 = 4\text{s}^{-1}, r_3 = 4\text{s}^{-1}$; energy consumed per measurement $e_1 = 3$J, $e_2 = 2.4$J, $e_3 = 2.1$J; and energy used for switching a sensor on and off $e_1^{\text{on}} = 10$J, $e_2^{\text{on}} = 8$J, $e_3^{on} = 5$J and $e_1^{\text{off}} = 2$J, $e_2^{\text{off}} = 1.5$J, $e_3^{\text{off}} = 1$J, respectively. Also, suppose that the current UUV configuration is $(x_1, x_2, x_3, sp) = (0, 1, 1, 2.8)$, and that sensor 3 experiences a degradation such that $r_3^{\text{new}} = 1\text{s}^{-1}$. The ENTRUST controller gets this new measurement rate through the monitor. As the sensor rates differ from those in the knowledge repository, the guard *analysisRequired()* returns true and the **startAnalysis!** signal is sent. Upon receiving the signal, the *analyser model* invokes the probabilistic verification engine, whose analysis results for requirements **R1–R3** are depicted in Fig. 17. The *analyse()* action filters the results as follows: configurations that violate requirements **R1** or **R2**, i.e., the shaded areas from Fig. 17a and Fig. 17b, respectively, are discarded.[9] The remaining configurations are feasible, so their cost (1) is computed for $w_1 = 1$ and $w_2 = 200$. The configuration minimising the cost (i.e., $(x_1, x_2, x_3, sp) = (1, 1, 0, 3.2)$ – circled in Fig. 17a-c) is selected as the best configuration. Since the best and the current configurations differ, the analyzer invokes the planner to assemble a stepwise reconfiguration plan with which i) sensor 1 is switched on; ii) next, sensor 3 is switched off; and iii) finally the speed is adjusted to 3.2m/s. Once the plan is assembled, the executor is enforcing this plan to the UUV system. The adaptation results from Fig. 17 provide the evidence required for the generation of the assurance case as described in Section 6.7.1.

9. Note that R1 and R2 are "conflicting" requirements, in the sense that the configurations that satisfy R1 by the widest margin violate R2, and the other way around. In such scenarios, ENTRUST supports the selection of configurations based on trade-offs between the conflicting requirements, as specified by a cost (or utility) function. If either requirement became much stricter (e.g. if R1 required over 50 measurements per every 10m), no configuration would satisfy both R1 and R2. In this case, ENTRUST would choose the configuration specified by the failsafe requirement R4, i.e. would reduce the UUV speed to 0m/s, and would record the probabilistic model checking evidence showing the lack of a suitable non-failsafe configuration.

6.6.2 FX System

In this stage, the self-adaptive FX system dynamically reconfigures in response to observed changes in the characteristics of the web services it uses. Several such reconfigurations are described later in the paper, in Section 7.1.2 and in Fig. 22. To illustrate this process in detail, consider the system configuration immediately after change C from Fig. 22, where the FX workflow uses the services MW_1, TA_0, FA_0, Al_0, Or_0 and No_1. This configuration is reached after the FX services, initially operating with the characteristics from Table 5, experience degradations in the reliability of MW_0 ($p_{\mathsf{MW}_0}^{\text{new}} = 0.9$, change B in Fig. 22) and in the response time of FA_1 ($time_{\mathsf{FA}_1}^{\text{new}} = 1.2$s, change C in Fig. 22). With the FX system in this configuration, suppose that the *Market Watch* service MW_0 recovers, i.e., $p_{\mathsf{MW}_0}^{\text{new}} = 0.976$ as in Table 5. Under these circumstances, which correspond to change D from Fig. 22, the ENTRUST controller receives the updated characteristics of MW_0 via its monitor. As the new service characteristics differ from those in the knowledge repository, the guard *analysisRequired()* holds and the **startAnalysis!** signal is sent. The analyser model receives the signal and invokes the runtime probabilistic verification engine, whose analysis of the FX requirements R1–R3 over the $2^6 = 64$ possible system

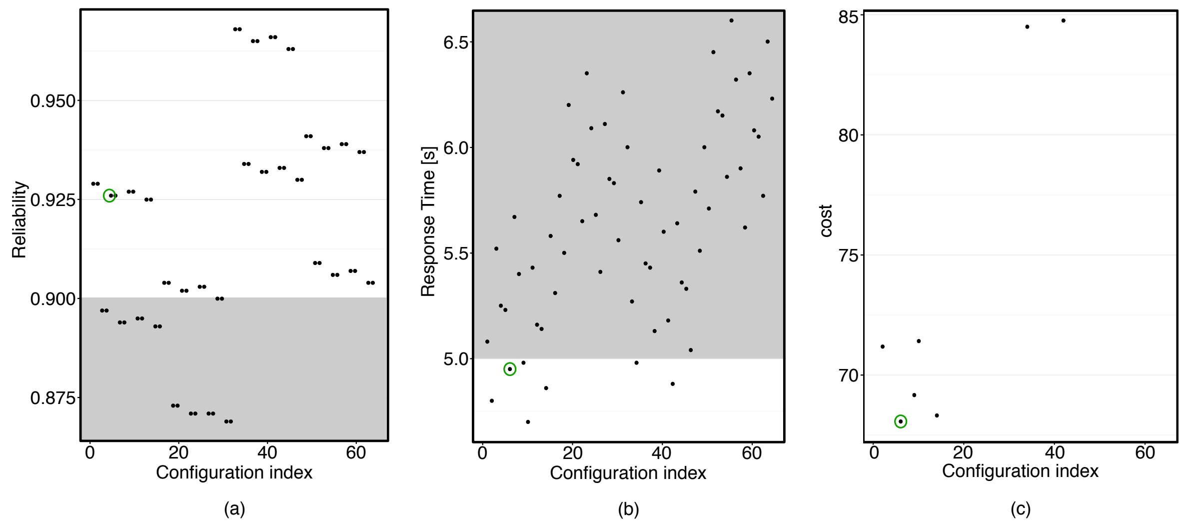

Fig. 18. Runtime verification results for FX requirement (a) R1, (b) R2, and (c) R3—cost of the feasible configurations, where the configuration index $i_1i_2i_3i_4i_5i_6$ in number base 2 corresponds to the FX configuration that uses services MW$_{i_1}$, TA$_{i_2}$, FA$_{i_3}$, AI$_{i_4}$, Or$_{i_5}$ and No$_{i_6}$. The best configuration (circled) has index $5_{(10)} = 000101_{(2)}$, corresponding to MW$_0$, TA$_0$, FA$_0$, AI$_1$, Or$_0$ and No$_1$. Shaded regions correspond to requirement violations.

configurations (corresponding to six services each provided by two implementations) is shown in Fig. 18. As part of this analysis, configurations that violate requirements R1 or R2 (i.e., those from the shaded areas in Fig. 18a and Fig. 18b, respectively) are discarded. The remaining configurations are feasible, so their cost is calculated (for $w_1 = 1$ and $w_2 = 2$) as shown in Fig. 18c. The feasible configuration using services MW$_0$, TA$_0$, FA$_0$, AI$_1$, Or$_0$ and No$_1$ has the lowest cost and is thus selected as the best system configuration. Since the best and the current configurations differ, the guard *adaptationRequired()* holds and the analyser invokes the planner through the **startPlanning!** signal to assemble a stepwise reconfiguration plan through which: (i) MW$_0$ replaces MW$_1$; and (ii) AI$_1$ replaces AI$_0$. Once the plan is ready, the executor automaton receives the **startExecuting?** signal and is ensuring the implementation of this plan by sending the signal **changeService!** to the system effectors.

6.6.3 Discussion

Both self-adaptive systems reconfigured in response to application-specific changes (more of which are described in Section 7). Selecting the new configurations involved the runtime probabilistic model checking of different types of stochastic models, to generate assurance evidence that system requirements were satisfied after each change.

6.7 Synthesis of Dynamic Assurance Argument

6.7.1 UUV System

In this stage, the partially-instantiated assurance argument pattern for the UUV system (Fig. 15) is fully instantiated after every selection of a new UUV configuration by the ENTRUST controller. For instance, after the ENTRUST controller activities described in Section 6.6.1 conclude with the selection of the UUV configuration $(x_1, x_2, x_3, sp) = (1, 1, 0, 3.2)$ and the generation of runtime verification evidence that this configuration satisfies requirements R1–R3, this partially-instantiated assurance argument pattern is fully instantiated as shown in Fig. 19.

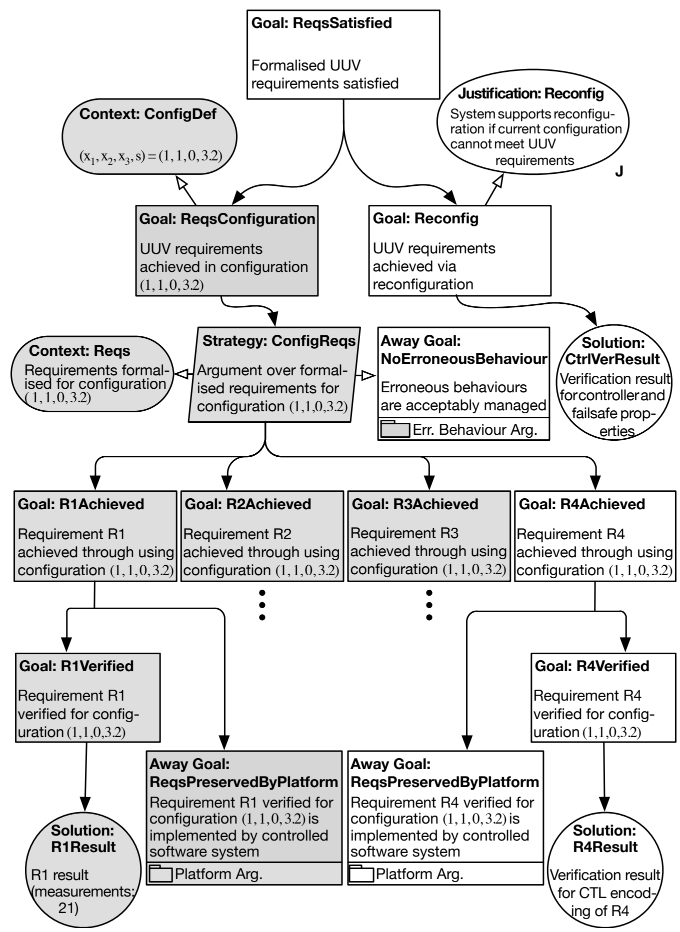

Fig. 19. Fully-instantiated assurance argument for the UUV system; the subgoals for **R2Achieved** and **R3Achieved** (not included due to space constraints) are similar to those for **R1Achieved**, and shading is used to show the elements instantiated at runtime

6.7.2 FX System

The partially instantiated FX assurance pattern from Fig. 16 is updated into a full assurance argument after each selection of a new configuration by the ENTRUST controller. This involves using the new evidence generated by the runtime probabilistic verification engine to complete the

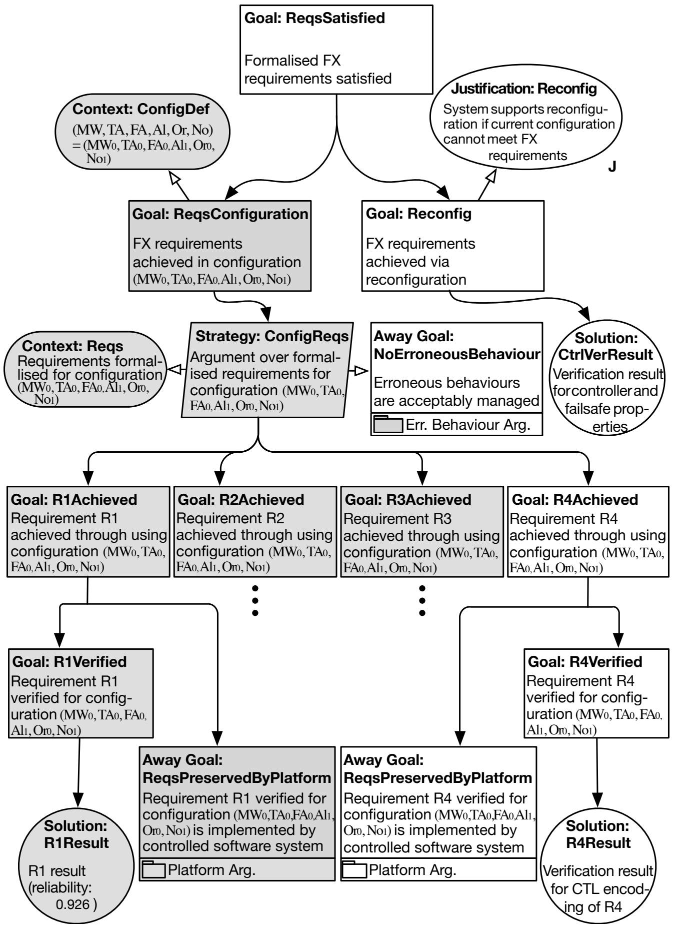

Fig. 20. Fully-instantiated assurance argument for the FX system; the subgoals for **R2Achieved** and **R3Achieved** (not included due to space constraints) are similar to those for **R1achieved**, and shading is used to show the elements instantiated at runtime

instantiation of the assurance pattern. As an example, Fig. 20 shows the complete assurance pattern synthesised as part of the configuration change that we used to illustrate the previous stage of ENTRUST in Section 6.6.2.

6.7.3 Discussion

For both the UUV system and the FX system, integrating the dynamically generated assurance evidence required the updating of only a few uninstantiated GSN '*solutions*' from the partially instantiated assurance arguments. In contrast, instantiating the current system configurations in the two assurance arguments involved multiple but small updates of '*context*', '*strategy*' and '*goal*' GSN elements. The rightmost branches of the assurance arguments ensure the goals associated with the failsafe requirements of the two systems, and therefore remained unchanged in this ENTRUST stage.

7 Evaluation

To evaluate the effectiveness and generality of ENTRUST, we used our methodology to engineer the self-adaptive software systems from Section 3. The two systems were developed as described in Section 6, and were deployed in a realistic environment seeded with simulated changes specific to their application domains. Finally, we examined the correctness and efficiency of the adaptation and of the assurance cases produced by ENTRUST in response to each of these unexpected environmental changes. The aim of our evaluation was to answer the following research questions.

RQ1 (Correctness): Are ENTRUST self-adaptive systems making the right adaptation decisions and generating valid assurance cases?

RQ2 (Efficiency): Does ENTRUST provide design-time and runtime assurance evidence with acceptable overheads for realistic system sizes?

RQ3 (Generality): Does ENTRUST support the development of self-adaptive software systems and dynamic assurance cases across application domains?

As the focus of our evaluation was the ENTRUST methodology and its tool-supported instance, we necessarily made a number of assumptions. In particular, we assumed that established assurance processes could be used to construct assurance arguments for all aspects of the controlled systems from our case studies, including their correct design, development, operation, ability to respond to effector requests, and any real-time considerations associated with achieving the new configurations decided by the ENTRUST controller. As such, these aspects are outside the scope of ENTRUST and are not covered in our evaluation. We further assumed that the derivation, validity, completeness and formalisation of the self-adaptive system requirements are addressed as part of the overall system assurance cases for the two case studies, and therefore also outside the scope of our evaluation of ENTRUST.

The experiments carried out to address the three research questions are described in Sections 7.1–7.3, and the main threats to validity are discussed in Section 7.4.

7.1 RQ1 (Correctness)

To answer the first research question, we carried out experiments that involved running the UUV and FX systems in realistic environments comprising (simulated) unexpected changes specific to their domains. For the UUV system, the experiments were seeded with failures including sudden degradation in the measurement rates of sensors and complete failures of sensors, and with recoveries from these problems. For the FX system, we considered variations in the response time and the probability of successful completion of third-party service invocation. All the experiments were run on a MacBook Pro with 2.5 GHz Intel Core i7 processor, and 16 GB 1600 MHz DDR3 RAM.

7.1.1 UUV System

For the UUV system, we described a concrete change scenario and the resulting self-adaptation process and generation of an assurance case in Sections 6.6.1 and 6.7.1. The complete set of change scenarios we used in this experiment is summarised in Fig. 21, which depicts the changes in the sensor rates and the new UUV configurations selected by the ENTRUST controller. The labels A–H from Fig. 21 correspond to following key events:

A) The UUV starts with the initial state and configuration from Section 6.6.1;

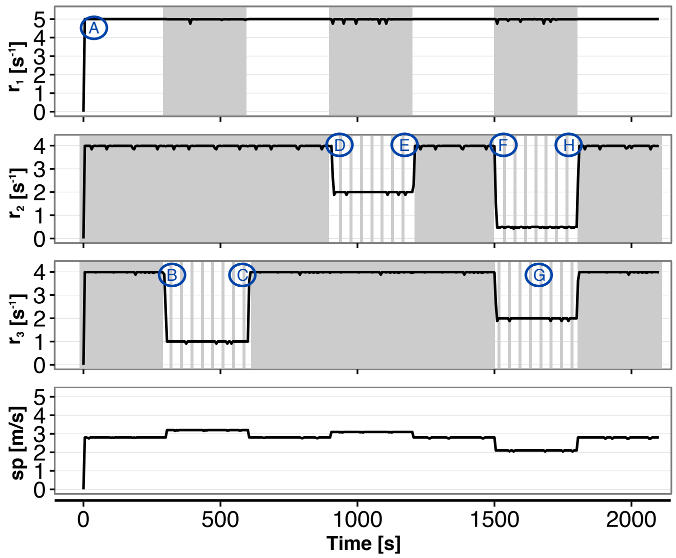

Fig. 21. Change scenarios for the self-adaptive UUV system over 2100 seconds of simulated time. Extended shaded regions indicate the sensors switched on at each point in time, and narrow shaded areas show the periodical testing of sensors switched off due to degradation (to detect their recovery).

B) Sensor 3 experiences the degradation described in Section 6.6.1 ($r_3^{\mathsf{new}} = 1$), so the higher-rate but less energy efficient sensor 1 is switched on (allowing a slight increase in speed to $sp = 3.2\mathrm{m/s}$) and sensor 3 is switched off;

C) Sensor 3 recovers and the initial configuration is resumed;

D) Sensor 2 experiences a degradation, and is replaced by sensor 1, with the speed increased to $sp = 3.1\mathrm{m/s}$;

E) Sensor 2 recovers and the initial configuration is resumed;

F) Both sensor 2 and sensor 3 experience degradations, so sensor 1 alone is used, with the UUV travelling at a lower speed $sp = 2.1\mathrm{m/s}$;

G) Periodic tests (which involve switching sensors 2 and 3 on for short periods of time) are carried out to detect a potential recovery of the degraded sensors;

H) Sensors 2 and 3 resume operation at nominal rates and the initial UUV configuration is reinstated.

If the UUV system was not self-adaptive, it would have to operate with a fixed configuration, which would lead to requirement violations for extended periods of time. To understand this drawback of a non-adaptive UUV, consider that its fixed configuration is chosen to coincide with the initial UUV configuration from Fig. 21 (i.e. $(x_1, x_2, x_3, sp) = (0, 1, 1, 2.8)$) – a natural choice because manual analysis can be used to find that this configuration satisfies the UUV requirements at deployment time. However, with this fixed configuration, the UUV will violate its throughput requirement R1 whenever one or both of UUV sensors 1 and 2 experience a non-trivial degradation, i.e. in the time intervals B–C (only 13 measurements per 10m instead of the required 20 measurements, according to additional analysis we carried out), D–E (only 15 measurements per 10m) and F–H (only 7 measurements per 10m) from Fig. 21. Although a different fixed configuration may always meet requirement R1, such a configuration would violate other requirement(s), e.g. having all three UUV sensors switched on meets R1 but violates the resource usage requirement R2 at all times.

Finally, we performed experiments to assess how the adaptation decisions may be affected by changes in the weights w_1, w_2 from the UUV cost (1) and the energy usage of the n UUV sensors. We considered UUVs with $n \in \{3, 4, 5, 6\}$ sensors, and for each value of n we carried out 30 independent experiments with the weights w_1, w_2 randomly drawn from the interval $[1, 500]$, and the energy consumption for taking a measurement and switching on and off a sensor (i.e., e_i, e_i^{on} and e_i^{off}, $1 \leq i \leq n$) randomly drawn from the interval $[0.1J, 10J]$. The experimental results (available, together with the PRISM-generated assurance evidence, on the project website) show that ENTRUST successfully reconfigured the system irrespective of the weight and energy usage values. In particular, if a configuration satisfying requirements R1–R3 existed for a specific change and system characteristics combination, ENTRUST reconfigured the UUV system to use this configuration. As expected, the configuration minimising the cost (1) depended both on the values of the weights w_1, w_2 and on the sensor energy usage. When no configuration satisfying requirements R1–R3 was available, ENTRUST employed the zero-speed failsafe configuration from requirement R4 until configurations satisfying requirements R1–R3 were again possible after a sensor recovery.

7.1.2 FX System

For the FX system, a concrete change scenario is detailed in Section 3.2, and the complete set of change scenarios used in our experiments is summarised in Fig. 22, where labels A–G correspond to the following events:

A) The FX starts with the initial services characteristics from Table 5 and uses a configuration comprising the services MW_0, TA_0, FA_0, AI_1, Or_0 and No_1, which satisfies requirements R1 and R2 and optimises R3;

B) The *Market Watch* service MW_0 experiences a significant reliability degradation ($p^{\mathsf{new}}_{\mathsf{MW}_0} = 0.9$), so FX starts using the significantly more reliable MW_1, and thus "affords" to also switch to the slightly less reliable but faster *Fundamental Analysis* service FA_1 in order to minimise the *cost* defined in requirement R3;

C) Due to an increase in response time of *Fundamental Analysis* service FA_1 ($time^{\mathsf{new}}_{\mathsf{FA}_1} = 1.2\mathrm{s}$), the FX switches to using FA_0 and also replaces the *Alarm* service AI_1 with the faster but more expensive service AI_0 (to meet the timing requirement R2);

D) The *Market Watch* service MW_0 recovers, so FX switches back to this services and also resumes using the less reliable *Alarm* service AI_1;

E) The *Technical Analysis* service TA_0 and the *Notification* service No_1 exhibit unexpected degradations in reliability ($p^{\mathsf{new}}_{\mathsf{TA}_0} = 0.98$) and in response time ($time^{\mathsf{new}}_{\mathsf{No}_1} = 1\mathrm{s}$), respectively, so the FX system self reconfigures to use MW_0, TA_1, FA_1, AI_0, Or_0 and No_0;

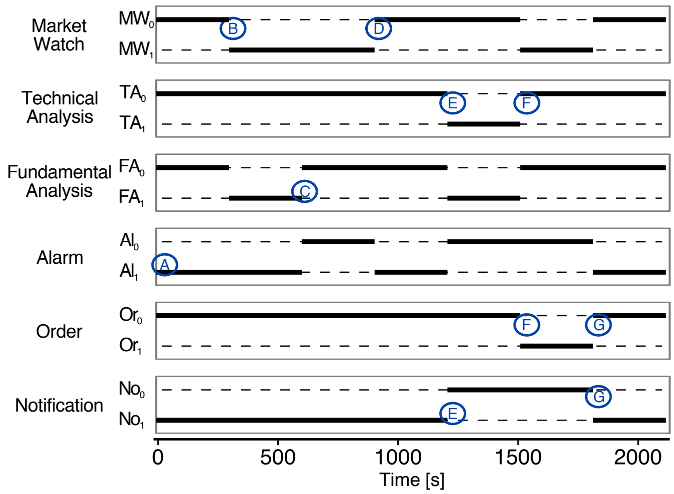

Fig. 22. Change scenarios for the self-adaptive FX system, with the initial services characteristics shown in Table 5. The thick continuous lines depict the services selected at each point in time.

F) As a result of a reliability degradation in the *Order* service Or_0 ($p^{\mathsf{new}}_{\mathsf{Or}_0} = 0.91$) and recovery of the *Technical Analysis* service TA_0, the FX system replaces services MW_0, TA_1, FA_1 and Or_0 with MW_1, TA_0, FA_0 and Or_1, respectively;

G) All the degraded services recover, so the initial configuration MW_0, TA_0, FA_0, Al_1, Or_0 and No_1 is reinstated.

As in the case of the UUV system, a non-adaptive FX version will fail to meet the system requirements for extended periods of time. For example, choosing to always use the initial FX configuration from Fig. 22 would lead to a violation of the reliability requirement R1 while service MW_0 experiences a significant reliability degradation in the time interval B–D. While using service MW_1 instead of MW_0 would avoid this violation, MW_1 is more expensive but no faster than MW_0 (cf. Table 5) so its choice would increase the cost (2), thus violating the cost requirement R3 in the time interval A–B.

7.1.3 Discussion

For each change scenario from our experiments within the two case studies (cf. Figs. 21 and 22), we performed two checks. For the former check, we confirmed that the ENTRUST controller operated correctly. To this end, we established that the change was accurately reported by the sensors and correctly processed by the monitor, leading the analyzer to select the right new configuration, for which a correct plan was built by the planner and implemented by the executor.

For the latter check, we determined the suitability of the ENTRUST assurance cases. We started from the guidelines set by safety and assurance standards, which highlight the importance of demonstrating, using available evidence, that an assurance argument is *compelling*, *structured* and *valid* [34], [81], [112]. Also, we considered the fact that ENTRUST has been examined experimentally but has not been tested in real-world scenarios to generate the industrial evidence necessary before approaching the relevant regulator. However, our preliminary results show, based on formal design-time and runtime evidence, that the primary claim of ENTRUST assurance cases is supported by a direct and robust argument. Firstly, the argument assures the achievement of the requirements either based on a particular active configuration or through reconfiguration, while maintaining a failsafe mechanism. Secondly, the argument and patterns are well-structured and conform to the GSN community standard [60]. Thirdly, ENTRUST provides rigorous assessments of validity not only at design time but also through-life, by means of monitoring and continuous verification that assess and challenge the validity of the assurance case based on actual operational data. This continuous assessment of validity is a core requirement for safety standards, as highlighted recently for medical devices [93]. As such, our approach satisfies five key principles of dynamic assurance cases [38]:

- *continuity* and *updatability*, as evidence is generated and updated at runtime to ensure the continuous validity of the assurance argument (e.g. the formal evidence for solution **R1Result** from the UUV argument in Fig. 19, which satisfies a system requirement given the current configuration);
- *proactivity*, since the assurance factors that provide the basis for the evidence in the assurance argument are proactively identified (e.g. the **ConfigDef** context from the UUV argument in Fig. 19, which captures the parameters of the current configuration);
- *automation*, because the runtime evidence is dynamically synthesised by the MAPE controller;
- *formality*, as the assurance arguments are formalised using the GSN standard.

In conclusion, subject to the limitations described above, our experiments provide strong empirical evidence that ENTRUST self-adaptive systems make the right adaptation decisions and generate valid assurance cases.

7.2 RQ2 (Efficiency)

To assess the efficiency of the ENTRUST generation of assurance evidence, we measured the CPU time taken by (i) the design-time UPPAAL model checking of the generic controller properties from Table 4; and (ii) the runtime probabilistic model checking performed by the ENTRUST analyzer. Fig. 23 shows the time taken to verify the generic controller properties from Table 4 for a three-sensor UUV system, and for an FX system comprising two third-party implementations for each workflow service. With typical CPU times of several minutes per property and a maximum below 12 minutes, the overheads for this design-time, once-only verification of all controller properties are entirely acceptable.

The CPU times required for the runtime probabilistic model checking of the QoS requirements for alternative configurations of the two systems (Fig. 24) have values below 1.5s and 2s, respectively. These runtime overheads, which correspond to under 10ms for the verification of a UUV configuration and under 30ms for the verification of an FX configuration, are acceptable because ENTRUST is intended for scenarios where:

1) failures and other changes requiring system reconfigurations are, on average, much less frequent than the

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSE.2017.2738640

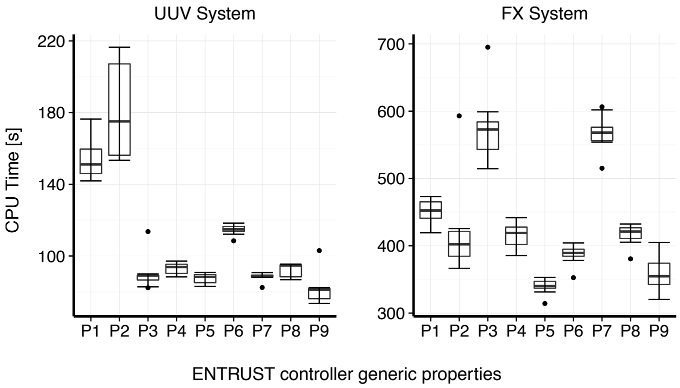

Fig. 23. CPU time for the UPPAAL verification of the generic controller properties in Table 4 (box plots of 10 independent measurements)

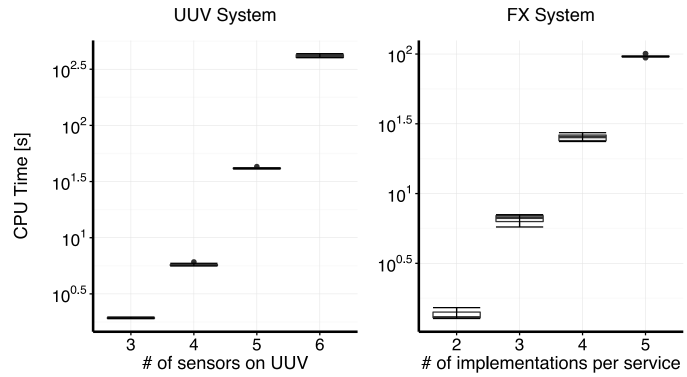

Fig. 24. CPU time for the runtime probabilistic model checking of the QoS requirements after changes (box plots based on 10 system runs comprising seven changes each—70 measurements in total)

frequency with which the runtime verification can be executed (i.e. every 1.5–2s for our two systems);

2) failsafe configurations can be temporarily assumed if needed during the infrequent reverifications of the ENTRUST stochastic models.

These assumptions ensure that, most of the time, ENTRUST adaptation decisions are reached before new changes occur and can be applied. They also ensure that any time spent in failsafe configurations is small compared to the time when the system employs "useful" configurations.

As shown in Fig. 24, we also ran experiments to assess the increase in runtime overhead with the system size and number of alternative configurations, by considering UUVs with up to six sensors, and FX system variants with up to five implementations per service. Typical for model checking, the CPU time increases exponentially with these system characteristics. This makes the current implementation of our ENTRUST instance suitable for self-adaptive systems with up to hundreds of configurations to analyse and select from at runtime. However, our recent work on compositional [21], incremental [70], caching-lookahead [55] and distributed [19] approaches to probabilistic model checking and on metaheuristics for probabilistic model synthesis [57] suggests that these more efficient model checking approaches could be used to extend the applicability of our ENTRUST instance to much larger configuration space sizes. As an example, in [55] we used *caching* of recent runtime probabilistic model checking results and anticipatory verification of likely future configurations (i.e. *lookahead*) to significantly reduce the mean time required to select new configurations for a variant of our self-adaptive UUV system (by over one order of magnitude in many scenarios). Integrating ENTRUST with these approaches is complementary to the purpose of this paper and represents future work.

7.3 RQ3 (Generality)

We used ENTRUST to develop an embedded system from the oceanic monitoring domain, and a service-based system from the exchange trade domain. As previously mentioned in Section 3 and summarised in Table 1, self-adaptation within these systems was underpinned by the verification of continuous- and discrete-time Markov chains, respectively; and the requirements and types of changes for the two systems differed. Finally, the ENTRUST assurance arguments for the two systems were based on assurance evidence obtained using multiple verification techniques:

1) testing evidence for the correct operation of trusted virtual machine;
2) model checking evidence for the correctness of the MAPE controller and the failsafe system requirements;
3) probabilistic model checking evidence for the remaining system requirements.

Although evaluation in additional areas is needed, these results indicate that our ENTRUST instance can be used across application domains.

To assess the overall generality of ENTRUST, we note that probabilistic model checking can effortlessly be replaced with simulation in our experiments, because the probabilistic model checker PRISM can be configured to use discrete-event simulation instead of model checking techniques. Using this PRISM configuration requires no change to the Markov models or probabilistic temporal logic properties we analysed at runtime. As for any simulation, the analysis results would be approximate, but would be obtained with lower overheads than those from Fig. 24.

The uncertainties that affect self-adaptive systems are often of a stochastic nature, and thus the use of stochastic models and probabilistic model checking to analyse the behaviour of these systems is very common (e.g. [15], [20], [27], [44], [47], [50], [90], [104]). As such, our ENTRUST instance is applicable to a broad class of self-adaptive systems.

Nevertheless, other methods have been used to synthesise MAPE controllers and to support their operation. Many such methods (e.g. based on formal proof, traditional model checking, other simulation techniques and testing) are described in Section 8. Given the generality of ENTRUST, these methods could potentially be employed at design time and/or at runtime by alternative instantiations of ENTRUST, supported by different modelling paradigms, requirement specification formalisms, and tools. For example, the use of the (non-probabilistic) graph transformation models or dynamic tests proposed in [6] and [51], respectively, in the self-adaptation ENTRUST stage is not precluded by any of our assumptions (cf. Section 4.2.1), although the method chosen for this stage will clearly constrain the types of requirements for which assurance evidence can be provided at runtime.

7.4 Threats to Validity

Construct validity threats may be due to the assumptions made when implementing our simple versions of the UUV and FX systems, and in the development of the stochastic models and requirements for these systems. To mitigate these threats, we implemented the two systems using the well-established UUV software platform MOOS-IvP and (for FX) standard Java web services deployed in Tomcat/Axis. The model and requirements for the UUV system are based on a validated case study that we are familiar with from previous work [55], and those for the FX system were developed in close collaboration with a foreign exchange expert.

Internal validity threats can originate from how the experiments were performed, and from bias in the interpretation of the results due to researcher subjectivity. To address these threats, we reported results over multiple independent runs; we worked with a team comprising experts in all the key areas of ENTRUST (self-adaptation, formal verification and assurance cases); and we made all experimental data and results publicly available to enable replication.

External validity threats may be due to the use of only two systems in our evaluation, and to the experimental evaluation having been done by only the authors' three research groups. To reduce the first threat, we selected systems from different domains with different requirements. The evaluation results show that ENTRUST supports the development of trustworthy self-adaptive solutions with assurance cases for the two different settings. To reduce the second threat, we based ENTRUST on input from, and needs identified by, the research community [25], [30], [36], [37]. In addition, we fine tuned ENTRUST based on feedback from industrial partners involved in the development of mission-critical self-adaptive systems, and these partners are now using our methodology in planning future engineering activities. Nevertheless, additional evaluation is required to confirm generality for domains with characteristics that differ from those in our evaluation (e.g., different timing patterns and types of requirements and disturbances) and usability by a larger number of users.

8 Related Work

Given the uncertain operating conditions of self-adaptive systems, a central aspect of providing assurances for such systems is to collect and integrate evidence that the requirements are satisfied during the entire lifetime. To this end, researchers from the area of self-adaptive systems have actively studied a wide variety of assurance methods and techniques applicable at design time and/or at runtime [30], [36], [84], [108], [119], [122], [128]. Tables 6 and 7 summarise the state of the art, partitioned into categories based on the main method used to provide assurances, e.g. formal proof, model checking or simulation. We consider as the main method of a study from our analysis the method that the study primarily focuses on; the approaches from these studies may implicitly use additional methods, such as testing of their platforms and tools, but this is not emphasised by their authors. We summarise the representative approaches included in each category according to their:

1) *Assurances evidence*, comprising separate parts for the methods used to provide assurance evidence for: (i) the correctness of the platform used to execute the controller, (ii) the correctness of the controller functions, and (iii) the correctness of the runtime adaptation decisions;
2) *Methodology*, comprising three parts: the engineering process (i.e. a methodical series of steps to provide the assurances), tool support (i.e., tools used by engineers to provide evidence at design time and tools used at runtime by the controller, e.g. during analysis or planning), and other reusable components (i.e. third-party libraries and purpose-built software components used as part of the controller, and other artefacts that can be used at design time or at runtime, including models, templates, patterns, algorithms).

Providing assurances for self-adaptive systems with strict requirements requires covering all these aspects, as well as an *assurance argument* that integrates the assurance evidence into a compelling, comprehensible and valid case that the system requirements are satisfied. Unlike ENTRUST (Table 8), the current research disregards this need for an assurance argument. We discuss below the different approaches and point out limitations that we overcome with ENTRUST.

Formal proof establishes theorems to prove properties of the controller or the system under adaptation. Proof was used to provide evidence for safety and liveness properties of self-adaptive systems with different semantics (one-point adaptation, overlap adaptation, and guided adaptation) [127]. Formal proof was also used to provide evidence for properties of automatically synthesised controllers, e.g. the completeness and soundness of synthesised behavioral models that satisfy an expressive subset of liveness properties [42] and correctness and deadlock free adaptations performed by automatically synthesised controllers [74]. Finally, formal proof was used to demonstrate the correctness of adaptation effects, e.g. proofs for safety, no deadlock, and no starvation of system processes as a result of adaptation [13], and guarantees for the required qualities of adaptations, e.g. proofs for optimised resource allocation, while satisfying quality of service constraints [1]. The focus of all these approaches is on providing assurance evidence for particular aspects of adaptation. All of them offer reusable components, however, these solutions require complete specifications of the system and its environment, and—unlike ENTRUST—cannot handle aspects of the managed system and its environment that are unknown until runtime.

Model checking enables verifying that a property holds for all reachable states of a system, either offline by engineers and/or online by the controller software. Model checking was used to ensure correctness of the adaptation functions that are modeled as interacting automata, with the verified models directly interpreted during execution by a thoroughly tested virtual machine [69]. Model checking was also used to provide guarantees for automatic controller synthesis and enactment, e.g. to assure that a synthesised controller and reusable model interpreter have no anomalies [12]. Model checking has extensively been used to provide guarantees for the effects of adaptation actions on the managed system, e.g. for safety properties of the transitions of a managed system that is modeled as

TABLE 6
Overview of related research on assurances for self-adaptive systems - part I

Approach	**Assurance evidence**			**Methodology**		
	Controller platform	**Controller functions**	**Adaptation decisions**	**Engineering process**	**Tool support**	**Other reusable components**
Formal proof						
Adaptation semantics [127]		Proof of safety and liveness properties of adaptive programs and program compositions				Model checking algorithm
Synthesis of behavioral models [42]		Proof of completeness and soundness of synthesized behavioral models				Controller synthesis technique
Controller synthesis [74]			Proof that controller synthesis algorithm generates controllers that guarantee correct and deadlock free adaptations	Controller synthesis process only	Tool to generate controller offline	Controller synthesis algorithm
Correctness adaptation effects [13]			Proof of safety, no deadlock, and no starvation of system processes as a result of adaptation			Verified middle-ware that ensures safety and liveness of monitored system
Guaranteed qualities [1]			Proof of optimizing resource allocation under QoS constraints			Ad-hoc solver of op-timisation problem
Model checking						
Correct adaptation functions [69]	Thoroughly tested virtual machine used to interpret and run controller models	UPPAAL model checking of interacting timed automata to ensure controller deadlock freeness, liveness, etc. and functional system requirements		UPPAAL used to verify controller models at design time		Tested reusable virtual machine; controller model templates
Controller synthesis and enactment [12]		Synthesised controller that is guaranteed not to be anomalous			Tool used for controller synthesis	Reusable inter-preter and config-uration manager for controller enactment
Safe adaptation configurations [6]			Verification of safety properties of system transitions using a graph transformation model			Symbolic verification procedure
Guaranteed qualities [18]			Probabilistic model checking of continually updated stochastic models of the controlled system and the environment to ensure non-functional requirements			PRISM verification library for analysis of stochastic system and envir-onment models
Resilience to controller failures [24]			Probabilistic model checking of resilience properties of synthesized Markov models of the managed system	Procedure to check resilience to controller failures		Reusable operational profiles to check resilience

a graph transformation system [6], to ensure non-functional requirements by runtime verification of continually updated stochastic models of the controlled system and the environment [18], and to provide evidence for resilience properties of synthesized Markov models of the managed system [24]. Again, the focus of all the approaches is on providing assurance evidence for particular aspects of adaptation. The ENTRUST instance presented in Section 5 uses two of these techniques (i.e., [69] and [18]) to verify the correctness of the MAPE logic at design time and to obtain evidence that adaptation decisions are correct at runtime, respectively. In addition, ENTRUST offers a process for the systematic engineering of all components of the self-adaptive system, which includes employing an industry-adopted standard for the formalization of assurance arguments.

Simulation approaches provide evidence by analysing the output of the execution of a model of the system. Simulation was used to evaluate novel self-adaptation approaches, e.g. to ensure the scalability and robustness to node failures and message loss of a self-assembly algorithm [106], and to support the design of self-adaptive systems, e.g., to check if the performance of a latency-aware adaptation algorithm falls within predicted bounds [27]. Recently some efforts have been made to let the controller exploit simulation at runtime to support analysis, e.g. runtime simulation of stochastic models of managed system and environment has been used to ensure non-functional requirements with certain level of confidence [121]. The primary focus of simulation approaches has been on providing assurance evidence for the adaptation actions (either as a means to check the controller effects or to make a prediction of the expected effects of different adaptation options). The approaches typically rely on established simulators.

Testing is a standard method for assessing if a software system performs as expected in a finite number of scenarios. Testing was used to test the effectiveness of adaptation frameworks, e.g. checking whether a self-repair framework applied to a client-server system keeps the latencies of clients within certain bounds when the network is overloaded [54]. Testing was used to provide evidence for the robustness of controllers by injecting invalid inputs at the controller's interface and use the responses to classify robustness [24]. Several studies have applied testing at runtime, e.g. to validate safe and correct adaptations of the managed system based on adapt test cases generated in response to changes in the system and environment [51]. While simulation and testing approaches can be employed within the generic ENTRUST methodology to obtain

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSE.2017.2738640

TABLE 7
Overview of related research on assurances for self-adaptive systems - part II

Approach	Assurance evidence: Controller platform	Assurance evidence: Controller functions	Assurance evidence: Adaptation decisions	Methodology: Engineering process	Methodology: Tool support	Methodology: Other reusable components
Simulation						
Evaluation novel approach [106]			Offline simulation to ensure the scalability and robustness to node failures and message loss			
Support for design [27]			Offline simulations to check if the performance of a latency-aware adaptation algorithm falls within predicted bounds		OMNeT++ simulator for checking algorithm performance	
Runtime analysis [121]			Runtime simulation of stochastic models of managed system and environment to ensure non-functional requirements with certain level of confidence		UPPAAL-SMC used for online simulation of stochastic system and environment models	
Testing						
Test effectiveness of adaptation framework [54]			Offline stress testing in client-server system, showing that self-repair significantly improves system performance			Rainbow framework to realise self-adaptation
Test controller robustness [24]		Robustness testing of controller by injecting invalid inputs at the controller's interface and employ responses to classify robustness		Robustness testing procedure only		Probabilistic response specification patterns for robustness testing
Runtime testing [51]			Dynamic tests to validate safe and correct adaptation of system using test cases adapted to changes in the system and environment	One-stage process for test case adaptation		
Other approaches						
Control-theoretic approaches, e.g., [49]		Control-theoretic guarantees for one goal (setpoint) using automatically synthesised controller at runtime	Controller guarantees for stability, overshoot, setting time and robustness of system operating under disturbances		ARPE tool to build online a first-order model of the system	Kalman filter and change point detection procedure for model updates
Runtime verification [103]			Online verification of the probability that a temporal property is satisfied given a sample execution trace		TRACE-CONTRACT tool used for trace analysis	
Sanity checks [116]			Sanity checks evaluate the correctness of resource sharing decisions made by a reasoning engine		CHAMELEON tool providing performance guarantees	

TABLE 8
Comparison of ENTRUST to related research on assurances for self-adaptive systems

Approach	Assurance evidence: Controller platform	Assurance evidence: Controller functions	Assurance evidence: Adaptation decisions	Assurance argument	Methodology: Engineering process	Methodology: Tool support	Methodology: Other reusable components
Generic ENTRUST methodology	Reuse of verified application-independent controller functionality	Verification of controller models to ensure generic controller requirements and some system requirements	Automated synthesis of adaptation assurance evidence during the analysis and planning steps of the MAPE control loop	Development of partial assurance argument at design time, and synthesis of dynamic assurance argument during self-adaptation	Seven-stage process for the systematic engineering of all components of the self-adaptive system, and of an assurance case arguing its suitability for the intended application	Tools specific to each ENTRUST instance	Reusable software artefacts: controller platform, controller model templates; Reusable assurance artefacts: platform assurance evidence, generic controller requirements, assurance argument pattern
Tool-supported ENTRUST instance	Reuse of thoroughly tested virtual machine to directly interpret and run controller models, and of established probabilistic model checking engine	UPPAAL model checking of interacting timed automata models to ensure controller deadlock-freeness, liveness, etc. and functional system requirements	PRISM probabilistic model checking of continually updated stochastic models of the controlled system and the environment to ensure non-functional requirements	Assurance argument synthesised using the industry-adopted Goal Structuring Notation (GSN) standard	Seven-stage process for the systematic engineering of all components of the self-adaptive system, and of an assurance case arguing its suitability for the intended application	UPPAAL used to verify controller models; PRISM used to verify stochastic system and environment models	Reusable controller platform (virtual machine, probabilistic verification engine), timed automata controller model templates; Reusable platform assurance evidence, CTL generic controller requirements, GSN assurance argument pattern

assurance evidence for particular aspects of self-adaptive systems, they need to be complemented by assurances for other components of a self-adaptive system and integrated in a systematic process as provided by ENTRUST.

Other approaches. We highlight some other related approaches that have been used to provide assurances for self-adaptive systems. Recently, there has been a growing interest in applying control theory to build "correct by construction" controllers [97]. The approach was used to automatically synthesise controllers at runtime, providing control-theoretic guarantees for stability, overshoot, setting time and robustness of system operating under disturbances [49]. Although promising, this research is at an early stage, and its potential to deliver solutions for real-world systems and scenarios has yet to be confirmed. In contrast, ENTRUST relies on proven software engineering techniques for modelling and analysing software systems and assuring their required properties. Runtime verification is a well-studied lightweight verification technique based on extracting information from a running system to detect whether certain properties are violated. For example, sequences of events can be modeled as observation sequences of a Hidden Markov Model allowing to verify the probability that a temporal property is satisfied by a run of a system given a sampled execution trace [103]. Sanity checks are another approach to check the conformance of requirements of adaptive systems. Sanity checks have been used to evaluate the correctness of resource sharing decisions made by a reasoning engine [116]. Approaches such as runtime verification and sanity checks are often supported by established tools. However, these approaches provide only one piece of evidence. Such approaches can also be used by our generic ENTRUST methodology, which supports the integration of assurance evidence from multiple sources in order to continuously generate an assurance case.

Another line of related research (not specifically targetting self-adaptation and thus not included in Table 7) is *runtime certification*, proposed in [94] and further developed in [75], [96]. Runtime certification involves the proactive runtime monitoring of the assumptions made in the assurance case, thereby providing early warnings for potential failures. ENTRUST goes beyond the mere monitoring of assumptions, to evolving the arguments and evidence dynamically based on the runtime verification data, particularly for self-adaptive software assurance. ENTRUST also extends existing work on assurance argument patterns [39] by enabling runtime instantiation.

The ENTRUST methodology and the other research summarised in this section also build on results from the areas of configurable software, configuration optimisation, and performance tuning. For instance, symbolic evaluation has been used to understand the behaviour of configurable software systems [92], dedicated support to automatically verify the correctness of dynamic updates of client-server systems has been proposed [66], and specification languages have been devised to help program library developers expose multiple variations of the same API using different algorithms [35]. Research in this area has been applied to realise self-adaptive software systems. For example, it has been used to deal with the problem of configurability of multi-tenant cloud settings by using a game theoretic approach that maximises tenants' preferences satisfaction [53], to find workarounds and add configuration guards to prevent particular failures [105], and to model the variability of cloud systems and identify reconfigurations that meet given criteria using temporal constraints and reconfiguration operations [99]. While these approaches address adaptation of highly configurable systems at runtime, they provide only specific pieces of evidence. Runtime testing as mentioned above (e.g. [51]) is one interesting approach to ensure that such systems continue to execute in a safe and correct manner when adapting to handle changing environmental conditions. Such an approach could be integrated in the generic ENTRUST methodology as part of the analysis phase to provide guarantees about the runtime decision making process of self-adaptation.

Finally, assurance cases and GSN in particular are related to goal modeling. Several approaches exist that provide alternative means to specify goal models for self-adaptive systems; we discuss a representative selection. RELAX offers a textual language that allows requirements to be temporarily relaxed to deal with uncertainty in adaptation [125]. RELAX has been integrated with traditional goal modeling using KAOS [29]. FLAGS provides both crisp goals specified in linear temporal logic and fuzzy goals specified in fuzzy temporal language [5]. Adaptations are triggered by violated goals and the goal model is modified accordingly to maintain a coherent view of the system and enforce adaptation on the running system. Other researchers specify requirements for adaptive systems as two complementary types: awareness requirements and evolution requirements [100]. Awareness requirements indicate the situations that require adaptation and evolution requirements prescribe what to do in these situations. The development of GSN was influenced by the research on goal modeling. Similar to the approaches discussed above, the notation is used to represent and decompose system goals, but in addition to that explicitly incorporate rationale arguments for the decomposition.

The sparsity of Tables 6 and 7 makes clear that existing approaches are confined to providing correctness evidence for specific aspects of the self-adaptive software. In contrast to existing work on assurances for self-adaptive systems, Table 8 shows that ENTRUST offers an end-to-end methodology for the development of trustworthy self-adaptive software systems. Unique to our approach, this includes the development of assurance arguments. The upper part of Table 8 shows how the generic ENTRUST methodology covers the whole spectrum of aspects that are required to provide assurances for self-adaptive systems with strict requirements. The lower part of Table 8 shows a concrete tool-supported instantiation of ENTRUST and summarises how the various assurances aspects are covered for this instance. Details about the information summarised in the table are provided in Sections 4 and 5.

9 Conclusion

We introduced ENTRUST, the first end-to-end methodology for the engineering of trustworthy self-adaptive software systems and the dynamic generation of their assurance cases. ENTRUST and its tool-supported instance presented in the paper include methods for the development of verifiable controllers for self-adaptive systems, for the generation of design-time and runtime assurance evidence, and for the runtime instantiation of an assurance argument pattern that we devised specifically for these systems.

The future research directions for our project include evaluating the usability of ENTRUST in a controlled experiment, extending the runtime model checking of system requirements to functional requirements, and reducing the runtime overheads by exploiting recent advances in probabilistic model checking at runtime [19], [21], [46], [55], [70]. In addition, we are planning to explore the applicability of ENTRUST to other systems and application domains.

References

[1] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian, "Resource management in the autonomic service-oriented architecture," in *2006 IEEE International Conference on Autonomic Computing*, June 2006, pp. 84–92.

[2] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, 1994.

[3] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, "Model-checking continuous-time Markov chains," *ACM Trans. on Computational Logic*, vol. 1, no. 1, pp. 162–170, 2000.

[4] C. Baier, B. R. Haverkort, H. Hermanns, and J. P. Katoen, "Model-checking algorithms for continuous-time Markov chains," *IEEE Trans. Softw. Eng.*, vol. 29, no. 6, pp. 524–541, 2003.

[5] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy goals for requirements-driven adaptation," in *18th IEEE International Requirements Engineering Conference*, 2010, pp. 125–134.

[6] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling, "Symbolic invariant verification for systems with dynamic structural adaptation," in *28th International Conference on Software Engineering*. ACM, 2006, pp. 72–81.

[7] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *QEST'06*, 2006, pp. 125–126.

[8] M. Benjamin, H. Schmidt, P. Newman, and J. Leonard, "Autonomy for unmanned marine vehicles with MOOS-IvP," in *Marine Robot Autonomy*. Springer, 2013, pp. 47–90.

[9] A. Bianco and L. de Alfaro, "Model checking of probabilistic and nondeterministic systems," in *FSTTCS'95*, 1995, pp. 499–513.

[10] P. Bishop and R. Bloomfield, "A methodology for safety case development," in *Industrial Perspectives of Safety-critical Systems*. Springer, 1998, pp. 194–203.

[11] R. Bloomfield and P. Bishop, "Safety and assurance cases: Past, present and possible future — an Adelard perspective," in *Making Systems Safer*. Springer, 2010, pp. 51–67.

[12] V. Braberman, N. D'Ippolito, N. Piterman, D. Sykes, and S. Uchitel, "Controller synthesis: From modelling to enactment," in *35th International Conference on Software Engineering*, 2013, pp. 1347–1350.

[13] O. Brukman, S. Dolev, and E. K. Kolodner, "A self-stabilizing autonomic recoverer for eventual byzantine software," *J. Syst. Softw.*, vol. 81, no. 12, pp. 2315–2327, Dec. 2008.

[14] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer, 2009, pp. 48–70.

[15] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic QoS management and optimization in service-based systems," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 387–409, 2011.

[16] R. Calinescu, K. Johnson, and Y. Rafiq, "Using observation ageing to improve Markovian model learning in QoS engineering," in *2nd ACM/SPEC Intl. Conf. on Performance Engineering*, 2011, pp. 505–510.

[17] ——, "Developing self-verifying service-based systems," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 734–737.

[18] R. Calinescu, "General-purpose autonomic computing," in *Autonomic Computing and Networking*, M. K. Denko, L. T. Yang, and Y. Zhang, Eds. Springer, 2009, pp. 3–30.

[19] R. Calinescu, S. Gerasimou, and A. Banks, "Self-adaptive software with decentralised control loops," in *FASE'15*, ser. LNCS. Springer, 2015, vol. 9033, pp. 235–251.

[20] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, Sep. 2012.

[21] 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*, ser. LNCS, vol. 7539. Springer, 2012, pp. 303–329.

[22] R. Calinescu and M. Z. Kwiatkowska, "Using quantitative analysis to implement autonomic IT systems," in *ICSE'09*, 2009, pp. 100–110.

[23] R. Calinescu, Y. Rafiq, K. Johnson, and M. E. Bakir, "Adaptive model learning for continual verification of non-functional properties," in *5th ACM/SPEC International Conference on Performance Engineering*, 2014, pp. 87–98.

[24] J. Camara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Robustness-driven resilience evaluation of self-adaptive software systems," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2015.

[25] J. Cámara, R. de Lemos, C. Ghezzi, and A. Lopes, Eds., *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7740.

[26] J. Cámara, D. Garlan, B. Schmerl, and A. Pandey, "Optimal planning for architecture-based self-adaptation via model checking of stochastic games," in *30th Annual ACM Symposium on Applied Computing*, 2015, pp. 428–435.

[27] J. Cámara, G. A. Moreno, D. Garlan, and B. Schmerl, "Analyzing latency-aware self-adaptation using stochastic games and simulations," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 10, no. 4, pp. 23:1–23:28, Jan. 2016.

[28] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis, "Automatic verification of competitive stochastic systems," in *18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, C. Flanagan and B. König, Eds. Springer, 2012, pp. 315–330.

[29] B. H. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *12th International Conference on Model Driven Engineering Languages and Systems*. Springer-Verlag, 2009, pp. 468–483.

[30] B. H. C. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. M. Villegas, "Using models at runtime to address assurance for self-adaptive systems," in *Models@run.time: Foundations, Applications, and Roadmaps*, N. Bencomo, R. France, B. H. C. Cheng, and U. Aßmann, Eds. Springer, 2014, pp. 101–136.

[31] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[32] E. Clarke, D. Long, and K. McMillan, "Compositional model checking," in *Proc. 4th Intl. Symp. Logic in Computer Science*, 1989, pp. 353–362.

[33] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003, pp. 331–346.

[34] Common Criteria Recognition Arrangement, "ISO/IEC 15408 – Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 4," September 2012.

[35] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active Harmony: Towards automated performance tuning," in *ACM/IEEE Conference on Supercomputing*, 2002, pp. 1–11.

[36] R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, "Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511)," *Dagstuhl Reports*, vol. 3, no. 12, pp. 67–96, 2014.

[37] R. de Lemos, H. Giese, H. A. Muller, M. Shaw *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013, vol. 7475, pp. 1–32.

[38] E. Denney, I. Habli, and G. Pai, "Dynamic safety cases for through-life safety assurance," in *ICSE'15*, 2015, pp. 587–590.

[39] E. Denney and G. Pai, "A formal basis for safety case patterns," in *Comp. Safety, Reliability, and Security*, ser. LNCS. Springer, 2013, vol. 8153, pp. 21–32.

[40] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: Multi-tier control for adaptive systems," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 688–699.

[41] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel, "Synthesis of live behaviour models for fallible domains," in *33rd International Conference on Software Engineering*, 2011, pp. 211–220.

[42] N. R. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel, "Synthesis of live behaviour models," in *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010, pp. 77–86.

[43] A. Elkhodary, N. Esfahani, and S. Malek, "FUSION: a framework for engineering self-tuning self-adaptive software systems," in *FSE'10*, 2010, pp. 7–16.

[44] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *31st International Conference on Software Engineering*, 2009, pp. 111–121.

[45] European Organisation for the Safety of Air Navigation, "Safety case development manual," 2006.

[46] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *ICSE'11*, 2011, pp. 341–350.

[47] ——, "A formal approach to adaptive software: continuous assurance of non-functional requirements," *Formal Asp. Comput.*, vol. 24, no. 2, pp. 163–186, 2012.

[48] A. Filieri, L. Grunske, and A. Leva, "Lightweight adaptive filtering for efficient learning and updating of probabilistic models," in *37th IEEE/ACM International Conference on Software Engineering*, 2015, pp. 200–211.

[49] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 299–310.

[50] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma, "Incremental runtime verification of probabilistic systems," in *Runtime Verification*, ser. LNCS, vol. 7687. Springer, 2012, pp. 314–319.

[51] E. M. Fredericks, B. DeVries, and B. H. C. Cheng, "Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty," in *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2014, pp. 17–26.

[52] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Quality prediction of service compositions through probabilistic model checking," in *Proc. 4th International Conference on the Quality of Software-Architectures, QoSA 2008*, ser. LNCS, S. Becker, F. Plasil, and R. Reussner, Eds., vol. 5281. Springer, 2008, pp. 119–134.

[53] J. García-Galán, L. Pasquale, P. Trinidad, and A. Ruiz-Cortés, "User-centric adaptation of multi-tenant services: Preference-based analysis for service reconfiguration," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS 2014. New York, NY, USA: ACM, 2014, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/2593929.2593930

[54] D. Garlan, S.-W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct 2004.

[55] S. Gerasimou, R. Calinescu, and A. Banks, "Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration," in *SEAMS'14*, 2014, pp. 115–124.

[56] S. Gerasimou, R. Calinescu, S. Shevtsov, and D. Weyns, "UNDERSEA: An exemplar for engineering self-adaptive unmanned underwater vehicles (artifact)," in *SEAMS'17*, 2017.

[57] S. Gerasimou, G. Tamburrelli, and R. Calinescu, "Search-based synthesis of probabilistic models for quality-of-service software engineering," in *30th Intl. Conf. Automated Softw. Eng. (ASE'15)*, 2015.

[58] L. Gherardi and N. Hochgeschwender, "RRA: Models and tools for robotics run-time adaptation," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, Sept 2015, pp. 1777–1784.

[59] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli, "Mining behavior models from user-intensive web applications," in *36th International Conference on Software Engineering*, 2014, pp. 277–287.

[60] GSN Working Group Online, "Goal structuring notation standard, version 1," November 2011.

[61] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[62] R. Hawkins, I. Habli, and T. Kelly, "The principles of software safety assurance," in *31st Intl. System Safety Conf.*, 2013.

[63] R. Hawkins, K. Clegg, R. Alexander, and T. Kelly, "Using a software safety argument pattern catalogue: Two case studies," in *Comp. Safety, Reliability, and Security*. Springer, 2011, pp. 185–198.

[64] R. Hawkins, I. Habli, and T. Kelly, "Principled construction of software safety cases," in *SAFECOMP 2013 Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems*, 2013.

[65] R. Hawkins, I. Habli, T. Kelly, and J. McDermid, "Assurance cases and prescriptive software safety certification: A comparative study," *Safety Science*, vol. 59, pp. 55–71, 2013.

[66] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *International Conference on Verified Software: Theories, Tools, Experiments*, ser. VSTTE'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 278–293. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27705-4_22

[67] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," 2001.

[68] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 1–28, 2008.

[69] M. U. Iftikhar and D. Weyns, "ActivFORMS: Active formal models for self-adaptation," in *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2014, pp. 125–134.

[70] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *16th Intl. ACM Sigsoft Symposium on Component-Based Software Engineering*, 2013, pp. 33–42.

[71] G. Karsai and J. Sztipanovits, "A model-based approach to self-adaptive software," *Intelligent Syst. and their Applications, IEEE*, vol. 14, no. 3, pp. 46–53, May 1999.

[72] T. Kelly and R. Weaver, "The goal structuring notation – a safety argument notation," in *Assurance Cases Workshop*, 2004.

[73] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, pp. 41–50, 2003.

[74] N. Khakpour, F. Arbab, and E. Rutten, "Synthesizing structural and behavioral control for reconfigurations in component-based systems," *Formal Aspects of Computing*, vol. 28, no. 1, pp. 21–43, 2016.

[75] J. Knight, J. Rowanhill, and J. Xiang, "A safety condition monitoring system," in *3rd Intl. Workshop on Assurance Cases for Softw. Intensive Syst.*, 2015.

[76] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 259–268.

[77] C. M. Krishna, *Real-Time Systems*. John Wiley & Sons, Inc., 2001.

[78] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, Part B, pp. 184–206, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157411921400162X

[79] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *CAV'11*, ser. LNCS, vol. 6806. Springer, 2011, pp. 585–591.

[80] M. Lahijanian, S. B. Andersson, and C. Belta, "Formal verification and synthesis for discrete-time stochastic systems," *IEEE Trans. Automat. Contr.*, vol. 60, no. 8, pp. 2031–2045, 2015.

[81] B. Littlewood and D. Wright, "The use of multilegged arguments to increase confidence in safety claims for software-based systems," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, pp. 347–365, 2007.

[82] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms*," *Real-Time Systems*, vol. 23, no. 1, pp. 85–126, 2002.

[83] F. D. Macías-Escrivá, R. Haber, R. del Toro, and V. Hernandez, "Self-adaptive systems: A survey of current approaches, research challenges and applications," *Expert Systems with Applications*, vol. 40, no. 18, pp. 7267–7279, 2013.

[84] J. Magee and T. Maibaum, "Towards specification, modelling and analysis of fault tolerance in self managed systems," in *SEAMS'06*, 2006, pp. 30–36.

[85] North European Functional Airspace Block, "NEFAB Project—Safety Case Report, Version 3.01," Dec. 2011.

[86] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Syst.*, vol. 14, no. 3, pp. 54–62, May 1999.

[87] D. Perez-Palacin, R. Calinescu, and J. Merseguer, "Log2Cloud: Log-based prediction of cost-performance trade-offs for cloud deployments," in *28th Annual ACM Symposium on Applied Computing*, 2013, pp. 397–404.

[88] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*, 1977, pp. 46–57.

[89] H. Psaier and S. Dustdar, "A survey on self-healing systems: approaches and systems," *Computing*, vol. 91, no. 1, pp. 43–73, 2011.

[90] T. Quatmann, C. Dehnert, N. Jansen, S. Junges, and J. Katoen, "Parameter synthesis for markov models: Faster than ever," in *14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2016, pp. 50–67.

[91] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, Jan 1989.

[92] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *ACM/IEEE International Conference on Software Engineering*. ACM, 2010, pp. 445–454.

[93] Royal Academy of Engineering, "Establishing High-Level Evidence for the Safety and Efficacy of Medical Devices and Systems," January 2013.

[94] J. Rushby, "The interpretation and evaluation of assurance cases," Comp. Science Laboratory, SRI International, Tech. Rep. SRI-CSL-15-01, 2015.

[95] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[96] D. Schneider and M. Trapp, "Conditional safety certification of open adaptive systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 8, no. 2, pp. 8:1–8:20, Jul. 2013.

[97] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Systematic literature review on control-theoretical software adaptation," *IEEE Trans. Softw. Eng.*, 2017.

[98] S. Shevtsov and D. Weyns, "Keep it simplex: Satisfying multiple goals with guarantees in control-based self-adaptive systems," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, 2016, pp. 229–241.

[99] G. Sousa, W. Rudametkin, and L. Duchien, "Extending dynamic software product lines with temporal constraints," in *12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2017.

[100] V. E. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos, "Requirements-driven software evolution," *Comput. Sci.*, vol. 28, no. 4, pp. 311–329, Nov. 2013.

[101] D. Spinellis, "Notable design patterns for domain specific languages," *Journal of Systems and Software*, vol. 56, no. 1, pp. 91–99, Feb. 2001.

[102] J. Spriggs, *GSN – The Goal Structuring Notation. A Structured Approach to Presenting Arguments*. Springer, 2012.

[103] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok, "Runtime verification with state estimation," in *Proceedings of the Second International Conference on Runtime Verification*, ser. RV'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 193–207.

[104] G. Su, T. Chen, Y. Feng, D. S. Rosenblum, and P. S. Thiagarajan, "An iterative decision-making scheme for Markov decision processes and its application to self-adaptive systems," in *19th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2016, pp. 269–286.

[105] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone, "Beyond the Rainbow: Self-adaptive failure avoidance in configurable systems," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 377–388.

[106] D. Sykes, J. Magee, and J. Kramer, "Flashmob: Distributed adaptive self-assembly," in *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 100–109.

[107] P. Tabuada, *Verification and Control of Hybrid Systems*. Springer, 2009.

[108] G. Tamura and et al., "Towards practical runtime verification and validation of self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013, vol. 7475.

[109] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, "A multi-agent systems approach to autonomic computing," in *Third Intl. Conf. on Autonomous Agents and Multiagent Syst.*, 2004, pp. 464–471.

[110] UK Civil Aviation Authority, "Unmanned aircraft system operations in UK airspace — Guidance. CAP 722. Sixth edition," 2015.

[111] UK Health & Safety Commission, "The use of computers in safety-critical applications," 1998.

[112] UK Ministry of Defence, "Defence Standard 00-56, Issue 4: Safety Management Requirements for Defence Systems," June 2007.

[113] UK Office for Nuclear Regulation, "The Purpose, Scope, and Content of Safety Cases, Rev. 3," July 2013.

[114] University of Virginia Dependability and Security Research Group, "Safety case repository," 2014.

[115] US Dept. Health and Human Services, Food and Drug Administration, "Infusion Pumps Total Product Life Cycle—Guidance for Industry & FDA Staff," 2014.

[116] S. Uttamchandani, L. Yin, G. A. Alvarez, J. Palmer, and G. A. Agha, "Chameleon: A self-evolving, fully-adaptive resource arbitrator for storage systems," in *USENIX Annual Technical Conference, General Track*, 2005, pp. 75–88.

[117] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas, "A framework for evaluating quality-driven self-adaptive software systems," in *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2011, pp. 80–89.

[118] W. Walsh, G. Tesauro, J. Kephart, and R. Das, "Utility functions in autonomic systems," in *IEEE International Conference on Autonomic Computing*, 2004, pp. 70–77.

[119] D. Weyns, N. Bencomo, R. Calinescu, J. Cámara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jezequel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli, "Perpetual Assurances in Self-Adaptive Systems," in *Software Engineering for Self-Adaptive Systems IV, Lecture Notes in Computer Science*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds. Springer, 2016.

[120] D. Weyns and R. Calinescu, "Tele Assistance: A self-adaptive service-based system examplar," in *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015, pp. 88–92.

[121] D. Weyns and M. U. Iftikhar, "Model-based simulation at runtime for self-adaptive systems," in *Proceedings of the 11th International Workshop on Models@Run.time*, ser. MODELS 2016, 2016.

[122] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad, "A survey of formal methods in self-adaptive systems," in *C3S2E'12*, 2012, pp. 67–79.

[123] D. Weyns, S. Malek, and J. Andersson, "FORMS: Unifying reference model for formal specification of distributed self-adaptive systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 1, p. 8, 2012.

[124] S. R. White, D. M. Chess, J. O. Kephart, J. E. Hanson, and I. Whalley, "An architectural approach to autonomic computing," in *Intl. Conf. on Autonomic Computing*. IEEE Computer Society, 2004, pp. 2–9.

[125] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J. M. Bruel, "Relax: Incorporating uncertainty into the specification of self-adaptive systems," in *17th IEEE International Requirements Engineering Conference*, Aug 2009, pp. 79–88.

[126] M. Zamani, N. van de Wouw, and R. Majumdar, "Backstepping controller synthesis and characterizations of incremental stability," *Systems & Control Letters*, vol. 62, no. 10, pp. 949–962, 2013.

[127] J. Zhang and B. H. Cheng, "Using temporal logic to specify adaptive program semantics," *Journal of Syst. and Softw.*, vol. 79, no. 10, pp. 1361 – 1369, 2006.

[128] P. Zoghi, M. Shtern, M. Litoiu, and H. Ghanbari, "Designing adaptive applications deployed on cloud environments," *ACM Trans. Auton. Adapt. Syst.*, vol. 10, no. 4, pp. 25:1–25:26, Jan. 2016.