The Journal of Supercomputing (2022) 78:18725–18753
https://doi.org/10.1007/s11227-022-04614-0

A distributed formal-based model for self-healing behaviors in autonomous systems: from failure detection to self-recovery

Imene Ben Hafaiedh[1] · **Maroua Ben Slimane**[2]

Accepted: 18 May 2022 / Published online: 13 June 2022

Abstract

The challenges of current software-intensive systems, large-scale information and computing systems environments, which are highly dynamic, heterogeneous, and unpredictable, have motivated the development of techniques that enhance these systems with autonomous behaviors. Even though different concerns about these systems have been deeply studied, their design is still considerably more challenging than traditional ones. Self-healing is one of the main features that characterize autonomic computing systems. Failure detection, recovery strategies, and reliability are of paramount importance to ensure continuous operation and correct functioning even in the presence of a given maximum amount of faulty components. Most existing research and implementations focus on architecture-specific solutions to introduce self-healing behaviors. This implies that users must tailor their software by taking into account architecture-specific fault tolerance features, which requires too much effort from developers and users. This paper proposes a distributed formal model for the specification, verification, and analysis of self-healing behaviors in autonomous systems, from failure-detection to self-recovery. Such a high-level model allows users to specify and apply the desired type of failure detection and recovery without requiring any knowledge about its implementation. Our model allows not only formal verification of different properties but also performance evaluation. We provide the verification of qualitative properties using state-space exploration tools, and quantitative properties are also validated through statistical model-checking. All these properties are preserved in actual implementation by ensuring that the deployed code is consistent with the validated model.

Keywords Autonomous systems · Formal models · Formal verification · Distributed systems · Failure-detection · Self-recovery

✉ Imene Ben Hafaiedh
imen.benhafaiedh@isi.utm.tn

Extended author information available on the last page of the article

1 Introduction

Autonomous systems are programmable systems that include dynamic evolutionary systems. They exhibit adaptive and anticipatory behavior, and they process not only data but also knowledge [1, 2]. Most of these systems are required to be capable of evolution and required to evolve dynamically every time new components are introduced and existing components are removed or fail. Autonomous systems include large-scale heterogeneous systems, embedded systems for automotive applications, telecommunications, wireless ad hoc systems, etc. [3–5]. Moreover, autonomous systems are software-intensive systems that are becoming increasingly complex and distributed. Such system complexity is rocketing beyond the ability to design, comprehend, and control as it approaches that of bio-systems. Indeed, in these systems, we do understand the behavior of components in isolation. However, we don't understand the global behavior of interaction components, which results in an imminent need for what we call "design for autonomy." Recently, autonomy design has gained more attention, and much research has been focusing on developing software-intensive systems by assembling autonomous and heterogeneous components capable of working together so as to ensure a set of objectives and quality goals in highly dynamic and unpredictable environments [6, 7]. Unfortunately, this leads to a highly complex design process that has to deal with unexpected events at run-time such as failure and attacks.

To deal with such highly dynamic and unpredictable environments, a desired functionality of autonomous systems is self-healing, which consists of detecting and managing systematically and dynamically failures and faults that have occurred [8]. The handling of such faults could be either simple by giving control of the system to a user or complex by applying predefined recovery strategies. Autonomous systems usually fall into the latter case, where they should implement, without any external intervention, complex recovery strategies that aim to bring the system back into a safe state [9]. Self-healing in autonomous systems represents an area of research that gathers increasing research interest but is still not very well studied in terms of scope, architectural models, and validation tools. Research related to self-healing behaviors has its origin in both fault-tolerant and self-stabilizing systems research. Fault-tolerant systems are characterized by their ability to handle transient and permanent failures in order to return to a safe state [10]. Self-stabilizing systems are considered a non-fault-masking approach for fault-tolerant systems [11].

Traditionally, to address the risks due to the overall design complexity of self-healing systems, researchers provide tools for test and simulation-based validation of execution scenarios [12]. Such tools are insufficient for guaranteeing at design time the requirements of applications and systems. To this end, a model-based design process relying on formal modeling semantics can offer the prospects for an exhaustive analysis of the application's behavior and a simulation analysis grounded on statistical confidence. Indeed, a promising approach to studying self-healing behaviors in autonomous systems is model-based development.

Models can serve as a vehicle for communicating information between control engineers and computer scientists. They provide a basis for the validation of different requirements at a pre-implementation level and for the automatic generation of source code.

In this context, we propose a model-based approach to study self-healing behaviors in autonomous systems. Our approach is based on formal semantics for system description and uses formal tools for performance analysis and formal verification. Our model defines an intelligent self-healing distributed model, as it dynamically and at run-time detects failures and selects an appropriate recovery strategy. If there is more than one component that needs to be healed, our model allows us to define priority policies over fault components. Such dynamic involvement is particularly required in complex decision-making environments.

Our particular interest is in distributed self-healing control, in which central control of failure detection and recovery is not an option. Indeed, today's autonomous systems have become highly distributed, necessitating the use of distributed control. Our main idea is to provide a high-level formal model in a completely distributed manner for the specification of self-healing systems, which allows us to formally verify different properties of such systems and to capture low-level details using stochastic and probabilistic modeling.

The aim of this work is to automatically analyze, study, and derive self-healing behaviors based on building correct components from the design of a system with faults, a set of reliability, availability, and safety requirements it must satisfy, and the recovery strategies to be applied in case of faults. We describe how to automatically define fault detection strategies by adapting a set of parameters in our proposed model to design the recovery strategies appropriate with respect to the given requirements. Moreover, it is highly advantageous for designers to automatically derive a correct-by-construction implementation from a high-level model. That is why our approach allows us to automatically generate distributed code for the validated behaviors. Such model implementations can be deployed with the system under study.

This paper presents a reasonable solution to the key issue, i.e., how to generate efficient and reliable failure detection and self-recovery protocols to facilitate the preliminary design of autonomous systems. To address this issue, we define a rigorous model-based design flow as one that guarantees essential system properties based on a distributed high-level formal model. This model is detailed enough to verify and execute low-level scenarios at the level of detail required to check key properties, and at the same time, it remains as simple and cost-effective as possible by focusing on what is required for checking the properties of interest. For this purpose, we advocate a design flow based on a component-based framework called behavior-interaction-priority (BIP) [13] and its stochastic version called SBIP [14, 15]. Both frameworks are based on a language with formally defined semantics for building executable models of mixed software and hardware systems. Such a combination allows us to handle both performance aspects and also functional behaviors such as timing constraints, which enables functional verification and performance evaluation to be performed in a consistent manner. We aim for a formal design approach that allows us to model important features of self-healing systems

at a level of abstraction suitable for applying different verification and analysis techniques, taking into consideration different low-level details in a probabilistic manner due to their significant variability. The main contributions can be highlighted as follows:

- We provide a high-level formal model for the description of failure detection and self-recovery behaviors in autonomous systems. Our model supports heterogeneity and scalability. Indeed, in our model, different component types can be defined and the number of resources can be automatically increased.
- Our model is completely distributed as we associate with each component a local monitor, which allows us to detect the components' failure occurrences. Then self-recovery strategies are performed through interactions between these monitors. Interactions are ensured with respect to a well-defined algebra of connectors offered by the formal semantics of the component-based framework BIP that has proved suitable for modeling and analyzing distributed systems.
- Our model integrates non-functional aspects along with the functional system behavior. It is stochastic as its components' behaviors are extended with stochastic features such as probabilities and distributions. Adding stochastic aspects permits model uncertainty in the design by including faults or execution platform assumptions. This allows us to additionally provide statistical model checking results for the system's behavior under assumptions about its external stimuli.
- To evaluate the applicability of our approach, first we modeled two well-known recovery strategies, namely migration and replication. Then we conducted a series of experiments to formally evaluate these strategies with respect to a set of required properties. Second, we applied our approach to a real-life case study for a robot, namely, the DALA robot [16].

The remainder of this paper is structured as follows. Section 3 illustrates the necessary background to understand the approach, in particular related to the semantics of the BIP framework. Section 2 motivates the paper by comparing it with related work. Section 4 describes the different components defining our proposed distributed model, and Sect. 5 details the different experimental results obtained with respect to the formal verification and performance analysis of different aspects of self-healing behaviors and requirements. We present in Sect. 6 a concrete application of the approach to a real-life case study for a module of the DALA Robot. Finally, in Sect. 7, we conclude with a summary and the challenges ahead.

2 Related work

Mastering the complexity of autonomous systems requires a combined effort of foundational research and new engineering techniques that are based on mathematically well-founded theories and approaches [17]. Thus, new proposed methods have to support the whole system life cycle, including requirements, design, implementation, maintenance, reconfiguration, and adaptation. Designing these systems and their components is usually accomplished through ad hoc processes, simulation

[18], and extensive testing to ensure their correctness [19]. A fault tolerance and self-recovery analysis is provided in [8]. Their analysis is based on a set of test sequences and focuses only on the hardware-architecture level addressing only hardware faults. However, in this work, we consider both hardware and software failure types. In [20], authors have proposed an approach based on testing models rather than operational systems, and thus raised the level of abstraction of testing from operational systems to models of their behaviors and properties. Their approach focuses on enabling software engineers to execute and validate a much larger number of test scenarios. It is based on the development of test strategies that select which scenarios should be executed on the deployed system depending on the level of risk they exhibit. Generally, these test methods fail to provide the desired confidence level, mainly due to the limited possible test cases to be analyzed with respect to the real state space [12, 21]. In our approach, we use formal methods, which are more powerful and provide mathematical techniques to create reliable systems.

In the context of fault detector design, a lot of effort has been dedicated to the design of optimal schedulers, providing efficient ways by which to render systems highly reliable [22]. In [23], a formal offline approach is used for the synthesis of an optimal scheduler scheme for dynamically arriving preemptive aperiodic task sets on multiprocessor systems that can tolerate permanent processor faults. Similarly, in [24], authors propose a supervisory control-based fault-tolerant scheduler synthesis scheme for real-time tasks modeled as precedence-constrained task graphs, executing on multicores. In both approaches, formal techniques are used to automatically synthesize optimal control logic for the task-nodes based on a global model of the system. Such centralized approaches, however, may not scale well and thus may be time-consuming and memory-intensive for large systems.

Unlike these research approaches, our architecture is not limited to multiprocessor architectures and it supports heterogeneity of components. It can be applied to study a variety of possible architectures for autonomous systems, which are architectures that represent a lot of heterogeneity between their components and where the requirements can change often and unexpectedly. The model we propose in this work allows designers to instantiate components with different parameters, thus allowing us to define a heterogeneous architecture. Moreover, we also allow designers to define hierarchical components that are not allowed in most of the existing offline synthesis approaches. In the existing research results related to fault detectors and containment approaches, [25, 26] proposed approaches are based on a centralized structure, which means that the information of all components is centralized in a node for processing and generating huge data volumes, which greatly increases the computational complexity. Such approaches propose methods that are unable to be widely used due to the limitation of communication bandwidth. In the same context of failure detection and containment, decentralized approaches have also been proposed, like the results obtained in [27, 28].

Such research results solve the problem of complexity in the centralized structure, but it does not take the information of other components into consideration, which makes the capability of fault detection not satisfying enough. The distributed approach, however, has gained considerable attention these days because of its good performance in the above two aspects. In our approach, we address a distributed

structure of failure detection and containment as the process of failure detection and self-recovery is ensured through a rich interaction model between a set of local monitors associated with each component of the system.

Formal methods have been considerably applied for the analysis of self-adaptive systems [29]. Most of the existing approaches are modeling a high-level of functionality, such as the autonomous management, control, evaluation, and organization of the overall system [30–32]. However, self-healing is defined as a system making, dynamically and by itself, all the necessary recovery steps to restore its distributed behavior to a specified mode of operations [33]. In other words, self-healing formal models and approaches have to focus on recovery as an elaborate process. Other works address adaptation and reconfiguration of autonomous systems and self-healing behavior by proposing architecture-based approaches [34–36]. Most of these works have described approaches assuming adaptation that can be specified and analyzed only at the architectural level. This limits the study to structural properties with a lack of formal analysis of the whole behavior of the system. Some architecture-based approaches to self-healing in autonomous systems use formal specification only for the description of architecture constraints that have to be met, which makes these approaches limited to structural properties and thus not enough general to open possibilities for functional, non-functional, or any behavioral property validation. In [37], based on architectural styles, the authors have developed an architecture-based approach to self-healing systems, which focuses more on detecting when to make a particular repair and choosing that repair. Their approach is based on a set of formally specified constraints over an architecture; a constraint violation is a reason for inducing a repair that is predefined in the system code. However, our approach does not require that the system make any assumptions about the types of repairs, and moreover, our model allows the designer to define any type of repair needed for the study. An architecture-based approach that handles behavioral aspects in addition to structural ones is proposed in [38]. They have defined SosADL, which is an ADL specially conceived for formally modeling the architecture of software-intensive systems-of-systems from both structural and behavioral viewpoints. Their architecture allows users to verify properties using the UPPAAL model checker. The use of such model-checking makes their approach not applicable in the case of large-scale complex systems because of the state space explosion problem. In our approach, we use a formal-based framework offering compositional verification [39] tools, which allows addressing the state space explosion problem inherent to model-checking timed systems with a large number of components.

In the context of existing model-based approaches to self-healing behaviors in autonomous systems, dynamic software architecture approaches have been introduced as a high-level view of the structural organization of systems. Most of these models are application-based, as in [7], where a specific formal framework is proposed for the verification of software-intensive space and aeronautical control applications, or in [40], where authors propose a performance analysis study specific to a Websphere Application Server.

In [41], authors address the formal verification of application-specific programs in designing and developing autonomous systems. Their approach is to automate the translation of programs, written in special purpose languages, into

the SMV model-checking language, perform model checking using standard algorithms, and then translate counter-examples back into terms that are meaningful to the software developer. Although their approach applies formal techniques to high-level languages that are geared toward autonomous systems and develops translators that produce code automatically. Their work is only limited to two languages, which makes their method not applicable when it comes to any other programming language. In [42], authors have proposed a formal model based on typed graph grammars extended with graph constraints and where the analysis and verification are performed using the AGG tool. However, their model allows only static analysis, which is relatively limited in the context of highly dynamic systems like self-healing ones.

Regarding our contribution, these existing model-based and formal approaches are limited to particular applications and thus cannot be applied to different architectures, which consequently limits the set of studied aspects related to failure detection and self-recovery. In our contribution, however, we propose a generic architecture with a parameterized model that allows us to describe different applications by adapting a set of parameters to design the behavior of the studied application. Hence, our research aims to provide a generic model compared to existing formal approaches and a completely distributed architecture compared to existing fault detectors and containment approaches.

3 Preliminaries

Our approach relies on the use of the behavior, interaction, priority (BIP) component framework [43] and its stochastic version (SBIP) [44]. BIP is a formal framework for building complex systems by coordinating the behavior of a set of atomic components. The behavior of each component is defined as a transition system extended with data and C/C++ functions. The coordination between components is layered, wherein the first layer consists of the component interactions, and the second layer involves dynamic priorities between interactions, which provides rich inter-component interaction features [45]. BIP also provides different tools for property verification and, in particular, automatic and distributed code generation. An atomic component is essentially a timed automaton labeled with ports used for communication among different components.

Definition 1 *(Atomic component)* An atomic component B is defined by the tuple $B = (L, P, C, T, \text{tpc})$ where L is a finite set of locations, P is a finite set of ports, C is a set of clocks, and $T \subseteq L \times (P \times G(C) \times 2^C) \times L$ is a set of transitions labeled with a port, a timing constraint and a subset of clocks to be reset. $\text{tpc} : L \longrightarrow G(C)$ assigns to each location $l \in L$ a time progress condition $tpc_l \in G(C)$. $G(C)$ is the set of timing constraints that are defined according to the following grammar: $\text{tc} := \text{true} \mid \text{false} \mid c \sim k \mid \text{tc} \wedge \text{tc}$, with $c \in C, k \in Z \geq 0$ and $\sim \in \{\leq, =, \geq\}$.

In practice, an atomic component can be extended with variables that are used to store local data. Moreover, each component transition can be associated with a Boolean condition specifying for which values of the local variables it is enabled, and an (internal) update function triggered along with transition execution which modifies the values of the variables.

A BIP model is built from a set of n atomic components:

$\{B_i = (L_i, P_i, C_i, T_i, \text{tpc}_i)\}_{i\in[1,n]}$, such that their respective sets of ports and clocks are pairwise disjoint.

Definition 2 *(Interaction)* An interaction between atomic components $\{B_i\}_{i=1}^n$ is a subset of ports $a \subseteq P$, such that it contains at most one port of every component, that is, $\mid a \cap P_i \mid \leq 1$ for all $i \in \{1, \ldots, n\}$. Since an interaction a uses at most one port of every component, we simply denote $a = \{p_i\}_{i\in I}$, where $I \subseteq \{1, \ldots, n\}$ and $p_i \in P_i$ for all $i \in I$. A component B_i is participating in a if $i \in I$.

Definition 3 *(BIP Model)* We denote by $B = \gamma(B_1, \ldots, B_n)$ the BIP model obtained by applying a set of interactions γ to the set of atomic components $\{B_i = (L_i, P_i, C_i, T_i, \text{tpc}_i)\}_{i=1}^n$. It is defined by the atomic component $B = (L, \gamma, C, T_\gamma, \text{tpc})$, where $L = L_1 \times \ldots \times L_n$, $C = \cup_{i=1}^n C_i$, $\text{tpc}(l) = \bigwedge_{i\in n} \text{tpc}_{l_i}$. A transition $\tau = (l, a, \text{tc}, r, l')$ from $l = (1_1, \ldots, l_n)$ to $l' = (1'_1, \ldots, l'_n)$ is in T_γ iff (1) $a = \{p_i\}_{i\in I} \in \gamma$, (2) for all $i \notin I$ $l'_i = l_i$ and (3) and there exist transitions $\tau_i = (l_i, p_i, \text{tc}_i, r_i, l'_i)$ of B_i, $i \in I$, such that $\text{tc} = \bigwedge_{i\in I} \text{tc}_i$, $r = \bigcup_{i\in I} r_i$.

In BIP, interactions are structured by *connectors*. A *connector* is a macro notation for representing sets of related interactions in a compact manner. To specify the set of interactions of a connector, two types of synchronizations are defined:

- strong synchronization or *rendez-vous*, when the only interaction of a connector is the maximal one, i.e., it contains all the ports of the connector.
- weak synchronization or *broadcast*, when interactions are all those containing any port initiating the broadcast.

Definition 4 *(Connector)* A connector Ω is defined as a tuple $(p_\Omega[x], \mathcal{P}_\Omega, \delta_\Omega)$ where:

- $p_\Omega[x]$ is a port called the *exported* port of Ω with x as associated variable. Such particular type of ports is used for building hierarchies of connectors.
- $\mathcal{P}_\Omega = \{p_i[x_i]\}_{i=1}^n$ is the set of connected ports called the *support set* of Ω. x_i is a variable associated with p_i.
- δ_Ω is the set of feasible interactions of Ω. $\forall a \in \delta_\Omega$, where $a = \{p_i\}_{i\in I}$ we define a tuple $(G_a, \mathcal{U}_a, \mathcal{D}_a)$ where,
  - G_a is a guard of a, an arbitrary predicate $G_a(\{x_i\}_{i\in I})$,
  - $\mathcal{U}_a$ is an upward update function of a of the form, $x := F^u(\{x_i\}_{i\in I})$,
  - $\mathcal{D}_a$ is a downward update function of a of the form, $\cup_{p_i}\{x_i := F^d_{x_i}(x)\}$.

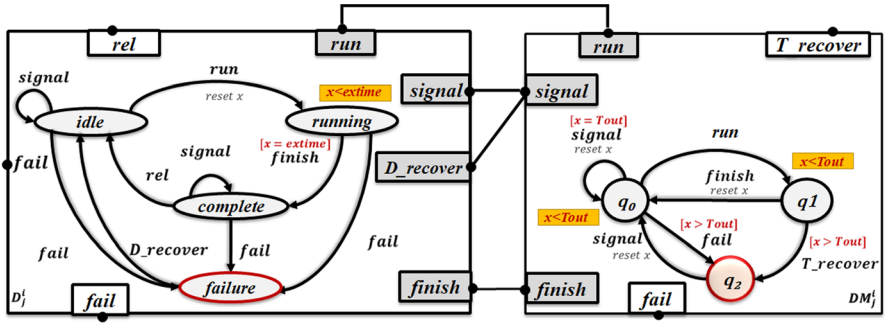

Fig. 1 Example behavior, interaction: device (left) and device monitor (right)

Figure 1 shows the composition of two BIP components from our model. On the right, the *Device-Monitor* observes the *Device* status. These components interact through the synchron ports *run*, *finish*, *D_recover* and *signal* using strict synchronization with (rendezvous) connectors. The *Device-Monitor* observes the execution of a task and its completion in the device through, respectively, the connectors $\{D.run, DM.run\}$ and $\{D.finish, DM.finish\}$ and it checks the device status temporary through the connectors $\{signal, signal\}$ and $\{D\_recover, signal\}$.

The stochastic extension of BIP [14] allows us to specify stochastic aspects of individual components and to provide a purely stochastic semantics for the parallel composition of components through interactions. Stochastic behavior at the level of atomic BIP components is obtained by using probabilistic variables. These are attached to probability distributions and are updated during transition firing, where they get random values accordingly. The semantics of transitions is thus fully stochastic. In our model, some components are specified using BIP formalism and extended with probabilistic aspects, in particular for the description of failure types.

4 Model description

In this section, we provide a detailed description of our approach based on a formal, distributed, and highly parameterized model, using the BIP formalism, which is a highly expressive, component-based framework with a rigorous semantical basis. Our model has a layered-based architecture Fig. 2, where each layer defines a particular type of component. Indeed, the overall architecture of our model is given as a network of BIP components interacting with each other using a rich interaction-model based on connectors. These connectors are memoryless links that allow one to define diverse interactions enriched with data exchange and priority rules [45]. In BIP, each component has a behavior and an interface. The behavior is a timed automaton and the interface is a set of ports allowing the corresponding component to communicate, interact and exchange data with the rest of the system (or with its environment). As we are addressing heterogeneous architectures, we define two *abstract* types of components,

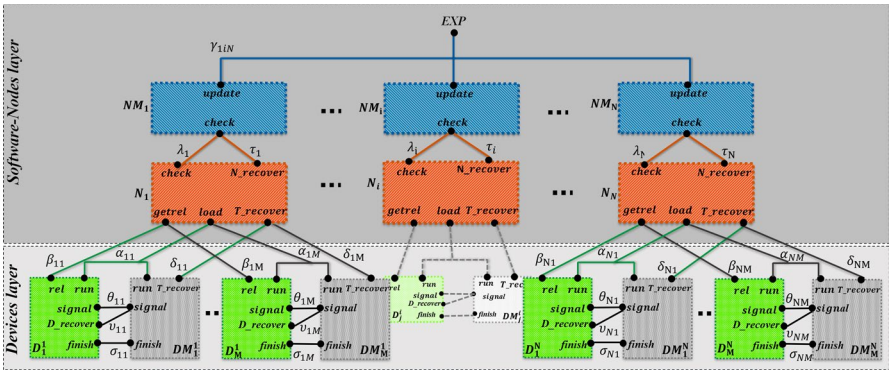

Fig. 2 Overall architecture of the proposed model

namely *Software-nodes* and *Devices*. We believe that any type of component in real autonomous systems could be naturally an instance of one of those two types. That is why we define an attribute *Type* for each component, allowing us to define a heterogeneous architecture.

- *Software-nodes*: host software elements that will be executed. They execute a set of input tasks-Queues by dispatching each task to the corresponding resources (cores, servers, buses, computing units,...). The detailed behavior of the software-node component is given in Fig. 3.
- *Devices*: are defined as both infrastructure and applicative entities. They can easily be instantiated to model a core, a sensor, an actuator or any computing unit. Their behavior is defined as the execution of a unique or a set of particular tasks assigned by their corresponding *Software-nodes*. Indeed, *Devices* provide resources to *Software-nodes*, on which functions, tasks, data treatment, or transfer could be performed. The behavior of a device is detailed in Fig. 3.

In our architecture, BIP connectors define some kind of logical binding, abstracting the communication models that allow software nodes and devices to interact and exchange data. In this work, as we are intending distributed architectures with a particular interest in distributed control of self-healing behavior, failure detection as well as the recovery process have to be performed in a distributed way. To this end, we associate with each *Software-node* and to each *Device* a local monitor called, respectively, *Node-Monitor* and *Device-Monitor* components (see Fig. 2). Such components monitor dynamically and at run-time their associated components in order to detect failures and then perform recovery. Failure detection can be done locally by monitors. However, the process of recovery needs communication between different monitors. This is ensured by the use of BIP connectors (see Table 1). These BIP connectors allow multiparty interactions, exchange of data, and updating of these data according to provided functions called *Update functions* associated with each interaction.

Table 1 Description of model's connectors

Connector/interaction	Description (update function)
$\{\alpha_{ij}\}_{i\in[1,N]}^{j\in[1,M]}\{load_{N_i}, run_{D_{ij}}, run_{DM_{ij}}\}_{i\in[1,N]}^{j\in[1,M]}$	Dispatches a task from the Node N_i to the device D_{ij} and the task data to its monitor DM_{ij} observing this interaction
$\{\beta_{ij}\}_{i\in[1,N]}^{j\in[1,M]}\{getrel_{N_i}, rel_{D_{ij}}\}_{i\in[1,N]}^{j\in[1,M]}$	Informs the node about the task completion and update the under execution queue of tasks (*QUnder*)
$\{\delta_{ij}\}_{i\in[1,N]}^{j\in[1,M]}\{T\_recover_{N_i}, T\_recover_{DM_{ij}}\}_{i\in[1,N]}^{j\in[1,M]}$	Transfers preempted tasks to node N_i whenever the device DM_{ij} fails
$\{\lambda_i\}_{i=1}^{N}\{check_{N_i}, check_{NM_i}\}_{i=1}^{N}$	Allows monitors to check and update their corresponding Nodes status
$\{\tau_i\}_{i=1}^{N}\{N\_recover_{N_i}, check_{NM_i}\}_{i=1}^{N}$	Allows the monitor to detect the node recovery and update the Node status
$\{\gamma_{ij}\}_{i,j=[1,N]}^{i\neq j}\{update_{NM_i}, update_{NM_j}\}_{i,j=[1,N]}^{i\neq j}$	Implements the recovery strategies (Replication, Migration) by invoking a C++ function detailed in Fig. 4)
$\{\sigma_{ij}\}_{i\in[1,N]}^{j\in[1,M]}\{finish_{D_{ij}}, finish_{DM_{ij}}\}_{i\in[1,N]}^{j\in[1,M]}$	Allows the monitor DM_{ij} to observe the task completion
$\{\theta_{ij}\}_{i\in[1,N]}^{j\in[1,M]}\{signal_{D_{ij}}, signal_{DM_{ij}}\}_{i\in[1,N]}^{j\in[1,M]}$	Allows the device-monitor to check the device status (failed or not)
$\{\vartheta_{ij}\}_{i\in[1,N]}^{j\in[1,M]}\{D\_recover_{D_{ij}}, signal_{DM_{ij}}\}_{i\in[1,N]}^{j\in[1,M]}$	Allows the device-monitor to detect the recovery of the device

4.1 Failure detection model

Failures are classified into two categories, namely applicative and infrastructure failures. In this work, we are considering both types. Failures of applicative entities are modeled as failures that affect software-nodes. A software-node crashes when it does not execute any further operations. For instance, infrastructure failures affect devices, and they can arise due to a hardware failure or a power failure. Note that an applicative failure can arise due to the infrastructure failures of its resources. In autonomous systems, several approaches to failure detection have been proposed. For applicative failures, our model integrates the most used technique, namely the *checkpointing* [46] (ping messages, heartbeats). Using this technique, the *Node-Monitor* component periodically checks the availability of the monitored component, by sending a message. The component is declared to have failed by its monitor, within a given timeout. Such a timeout is defined as an invariant in the behavior of the *Node-Monitor*. To avoid false failure detection, our model integrates a technique that allows the monitor to define a number of attempts over which the component is declared as failed. Note that the timeout and the number of attempts are parameters of the monitor components, which makes it possible to choose their appropriate values with respect to the application under study (see Table 2). The detailed behavior of the *Node-Monitor* is given in Fig. 3.

For the monitoring of *devices*, which means the detection of infrastructure failures, *messages observation* technique is preferred. The monitor observes the functional messages of the device. For example, if the latter communicates at regular intervals, there is no impact on the monitored device. A device failure is declared

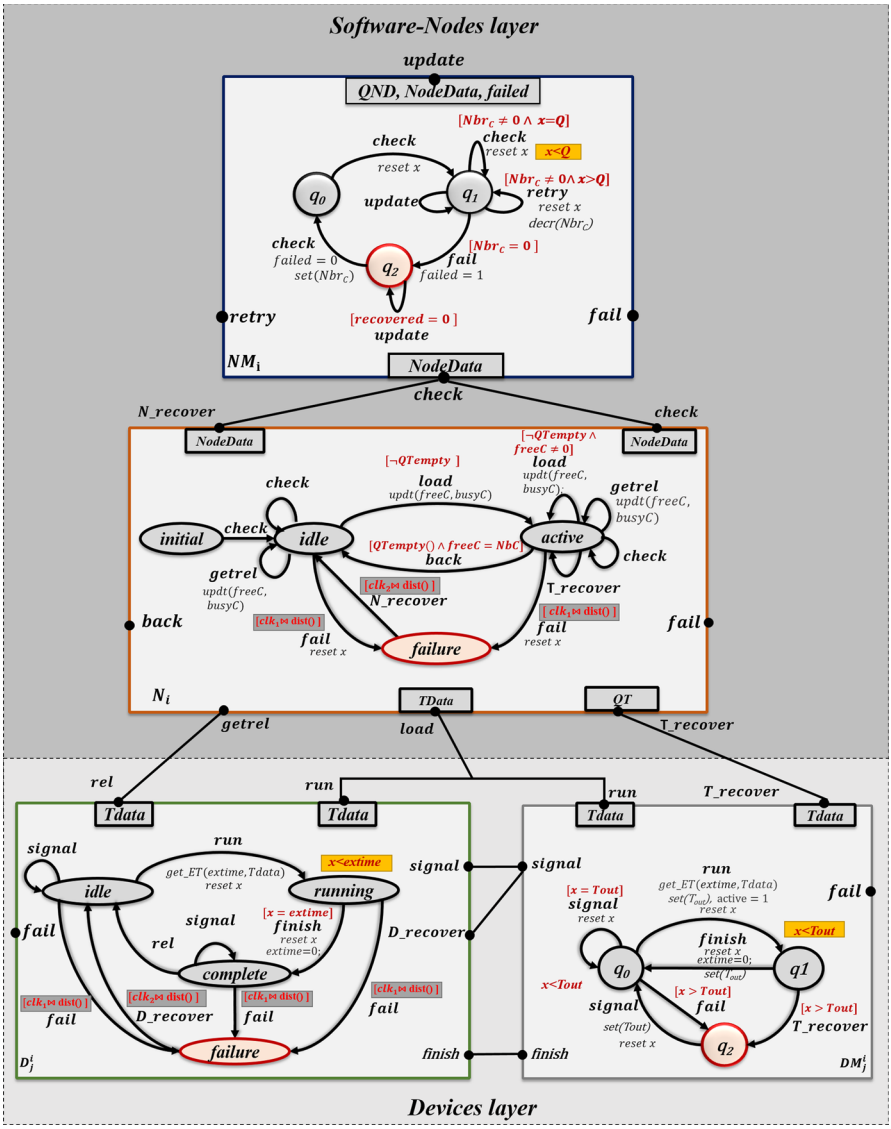

Fig. 3 Software-nodes N_i, node-monitor NM_i, device D_i and device-monitor DM_i components

when the monitor fails to observe a message at the required interval. The detailed behavior of the *Device-Monitor* components is given in Fig. 3. It is connected to the monitored device as well as to the *Software-node* to which the device provides services (data, computation...). The monitor observes functional messages of the devices, namely the *load* and *finish* transitions. When the device is in an *idle* state (with no executions to perform), it periodically sends a liveness signal to its monitor.

In the case of failure detection, the corresponding *Software-node* will be informed through the set of connectors $\{\delta_{ij}\}_{i\in[1,N]}^{j\in[1,M]}$ (see Table 1).

4.2 Self-recovery model

In self-healing behaviors, and after failure detection, a recovery process has to take place. Different recovery strategies have been proposed in the context of self-healing systems. The most commonly used techniques are replication (redundancy) strategies [47] and task migration (reassignment) strategies. Our model integrates both strategies and allows for easy integration of newly proposed ones. The recovery strategy is implemented as an update function over the set of connectors $\{\gamma_{ij}\}_{i=1\neq j}^{N}$ (see Table 1). Such connectors allow collecting data from all connected monitors and thus applying the needed recovery strategy through a C++ function detailed in Fig. 4. Indeed, the *recover_function* takes as input the queue of node data containing updated data for each node at a given time. Failed nodes are stored in a queue of node data. For each failed node, a queue of active nodes of the same type is defined. Then, the adequate recovery strategy is executed depending on the returned results of *replication_test* and *migration_test* functions. The *replication_test* function checks if there is at least one active node executing the same queue as the failed node. In that case, the failed node is recovered using a replication strategy. If the recovery using replication is not possible, the *migration_test* function checks if the migration strategy can be applied or not. If it returns false, then the failed node waits until it recovers, else the queue of tasks of the failed node will be equally distributed among active nodes.

In addition to recovery strategies implemented using monitors at the control level, low-level self-recovery can also be possible. Indeed, a device or a node can recover within a given time (self-repair). This is modeled in each component's behavior using the transition *recover*, allowing them to return to the active state of both the *Software-node* and the *Device* (see Fig. 3).

In this work, we intend to model the real variability of different parameters and behaviors. For this purpose, in our model we combine the use of BIP and its statistical version, namely SBIP. Stochastic semantics are integrated into our model by adapting its deterministic semantics with probabilistic variables. This leads to behaviors that combine both stochastic and non-deterministic aspects, leading to a more expressive model. In particular, at the level of the failure or recovery transitions of the behaviors of nodes as well as devices, any probability or any probabilistic distribution could be defined. Indeed, stochastic interactions could be defined by tagging component ports with S-BIP specific annotations. In this context, and in our model, the set of ports $\{fail, N\_recover, D\_recover\}$ are tagged by probability density functions and their parameters, which allows us to associate any distribution to specify different variants of failures or recovery behaviors (see Fig. 3).

This model is general enough to be applied to other failure detection and recovery strategies. The correctness of this model is proven using BIP tools, and the results are presented in the following section. In addition, results of the verification of different requirements and properties related to the strategies implemented are also presented.

Table 2 Parameters and variables description

Parameters and variable	Description
N, M_i, QND	Number of nodes, Number of devices associated with a Node i, Queue of N Nodes (NodeData)
$NodeData_i$	Structure containing: $idNode_i$(id of the Node), MyQ_i(Queue of Tasks), $idQueue_i$ (the id of MyQ_i), $active_i$(Node is active or not), $type_i$(type of the Node), $QUnder_i$(under execution tasks.) and $Qtoaccept_i$(accepted tasks by migration)
T_{out}	Interval of time given to a node to answer as alive before the number of chances is decreased
Nbr_C	Number of chances given to a node to be declared as failed
$Device_j$	Structure containing: $idDevice_j$(id of the Device), $active_j$(Device is active or not), T_j (Task under execution)
Q	Interval of time given to a device to answer as alive before it is declared as failed
T_data	Structure containing the following *Task* attributes: *idtask*(Task id), *extime*(execution Time) and $exec_T$(Under execution or not)

5 Performance analysis and formal verification

The second contribution of this article concerns formal verification and performance analysis. Traditionally, performance models separated from the functional behavior of the system are built late, and then only simulation or analytic techniques are used for analysis. We propose combining formal verification of functional self-healing behaviors using BIP together with statistical guarantees provided by its statistical version. Combining both frameworks is highly motivated by the idea of encompassing functional and performance aspects at an appropriate level of abstraction. All experiments have been run on an Intel Core *i*7 running at 2.40 GHz with a 64-bit Linux operating system and 16 GB of RAM.

5.1 Invariance and stochastic verification

Autonomous behaviors are prone to event scheduling delays. That is why *Invariance* properties are very interesting when it comes to real-time systems in general. Indeed, such properties guarantee the respect of a set of timing constraints associated with system states. For our model, the following invariant requirements were formalized as qualitative properties that were checked using the RTD-Finder tool [39]:

1. $Inv_i^1 = (x_i < Q_i)$ is defined in the state q_1 of each *Node-Monitor* NM_i. Such invariant ensures that the status of the controlled Node N_i is checked at each period Q_i and it is formally described using Temporal-Logic as follows: $\forall i \in [1, N]$, $AG\ (q_{1_i} \Longrightarrow (x_i \leqslant Q_i))$

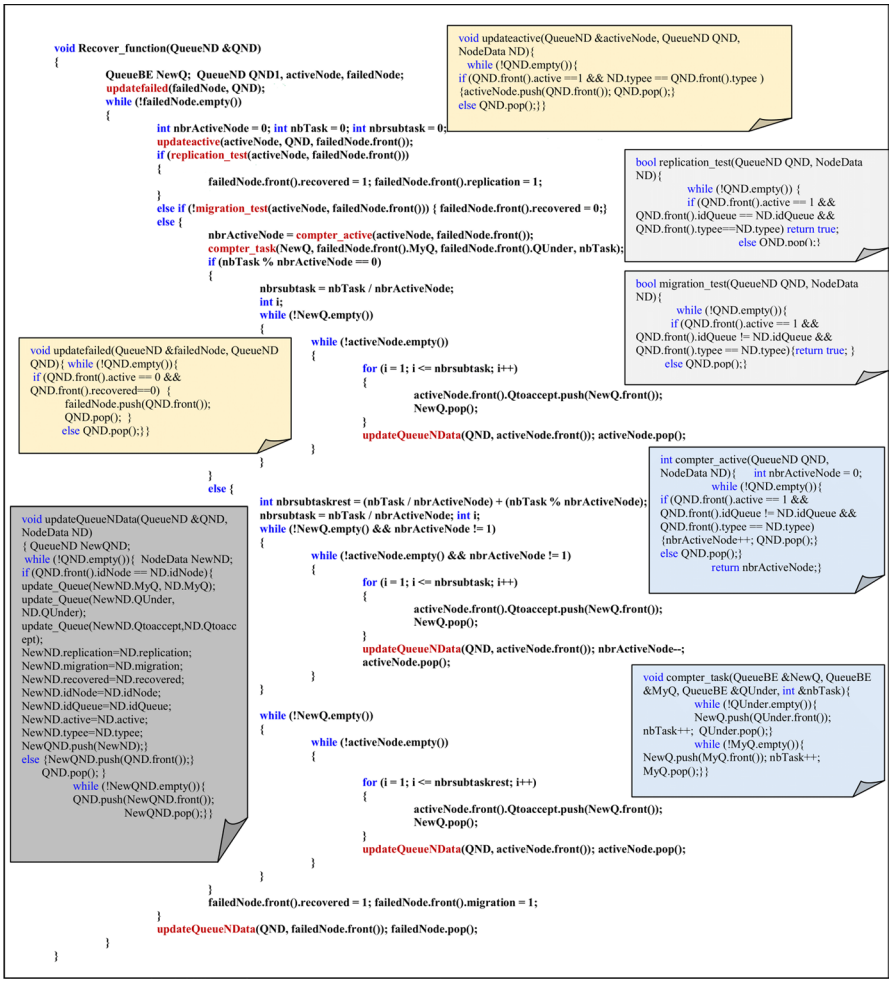

Fig. 4 Recovery strategy function

2. $Inv_{ij}^{2} = (x_{ij} \leqslant extime_{ij})$ is defined at the state $running_{ij}$ of the *Device* D_j^i. The satisfaction of this invariant guarantees that D_j^i cannot stay in this state more than the execution time *extime* of the loaded task. This invariant is formally described as follows: $\forall j \in [1, M], i \in [1, N]\, AG\,(running_{ij} \Longrightarrow (x_{ij} \leqslant extime_{ij}))$
3. $Inv_{ij}^{3} = (x_{ij} \leq T_{outij})$ is defined at the state $q_{0_{ij}}$ (and $q_{1_{ij}}$) of each *Device-Monitor* DM_j^i to guarantee that the state of the controlled *Device* is checked at each period T_out. This invariant property is formally described using Temporal-Logic as follows $\forall j \in [1, M], i \in [1, N]$: $AG(q_{0_{ij}} \Longrightarrow (x_{ij} \leqslant T_{outij}))\, AG(q_{1_{ij}} \Longrightarrow (x_{ij} \leqslant T_{outij}))$

Figure 5 gives the obtained results related to the verification of the set of predefined invariants. Our model is parametric, with N (Nodes) and M (Devices). In particular, we measure in Fig. 5a, the verification time of these invariants when making the model more complex by increasing the number of components. We have considered different topologies up to a total number of 1200 components (Nodes, Devices, and Monitors). Such results give an idea of the overhead introduced by our model, as well as the scalability of our approach model and its capability to model real architectures (see Fig. 5b). Once our model is formally verified for guaranteeing behavioral correctness properties (invariance), essential quantitative properties are also validated through statistical model checking of a stochastic BIP. Statistical model checking is a simulation-based analysis with statistical guarantees, i.e., finely controlled quality of results by various confidence parameters. It was proposed as a means to cope with the scalability issues in numerical methods for the analysis of stochastic systems [14]. The SMC of our model is automated by the SMC-BIP tool that supports all types of MTL properties and accepts, as inputs, the property; our model in SBIP; and the confidence parameters α, β, and δ. It provides a verdict in the form of the probability of the property's holding true. In the experimentation performed in our model, we checked the following set of MTL properties:

- The ith *Node-Monitor* eventually declares its corresponding *Node* as failed, within a given time, after the *Node* effectively fails. $P_1(t) \equiv \Diamond_{[0,t]} Monitor_i.failed \vee \Box_{[0,t]} !Node.Failed$
- The ith *Device-Monitor* eventually declares its corresponding *Device* as failed, within a given time, after the *Device* effectively fails. $P_2(t) \equiv \Diamond_{[0,t]} Monitor_i.failed \vee \Box_{[0,t]} !Device.Failed$
- The ith *Node* will eventually recover after failure. $P_3(t) \equiv \Diamond_{[0,t]} Node_i.Recover \vee \Box_{[0,t]} !Node.Failed$
- The ith *Device* will eventually recover after failure. $P_4(t) \equiv \Diamond_{[0,t]} Device_i.Recover \vee \Box_{[0,t]} !Device.Failed$

Figure 6a–d summarizes the results obtained by applying the probability estimation algorithm implemented in SBIP, with a confidence of 10^{-1} and a precision of 10^{-1}. We have applied the algorithm for each defined property for the interval [0, t], with a value of t equal to 500ms. We investigated different topologies' impacts (18, 30, 60, 90, and 120 components) on the probability of

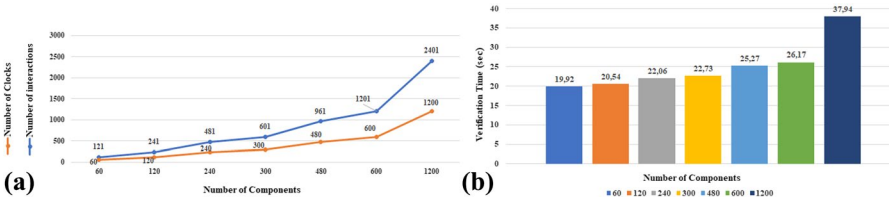

Fig. 5 Formal verification results

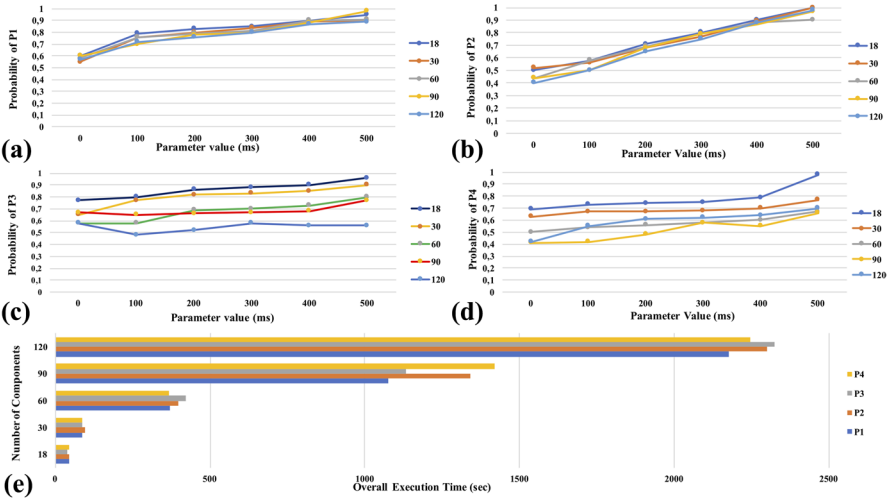

Fig. 6 **a** Probability of P_1, **b** probability of P_2, **c** probability of P_3 and **d** probability of P_4. **e** The overall execution time to compute probability for each property when increasing the number of components

the different properties. We can observe that the number of components of the system does not have an important impact on the probability of P_1 and P_2 as both properties are related to the behavior of each monitor with respect to each monitored component. However, this is not the case for the properties P_3 and P_4 because they depend on the recovery strategy, which may involve the rest of the system. Figure 6e illustrates the overall execution time taken by SBIP to estimate the probability for each property when increasing the total number of components.

5.2 Simulation analysis

More experiments and simulation analysis are also possible using the automatically generated code of our formal model. This generated code is proven to be *semantically equivalent* to the initial model, which means that any properties already checked on the model still hold on its generated code. Indeed, using the executable code generated by the real-time BIP, real-time executions are run and different sets of experiments can be observed. We have derived a set of experiments to study the different self-healing strategies as well as their impact on the system behavior and performance when varying different parameters of our model.

Experiment 1 In this experiment, we aim to study the impact of both the percentage of failed nodes and the recover time on the number of accomplished tasks within a given period of time. Our study is based on the configuration detailed in Fig. 7 and focuses on self-repair systems. Remember that in our model, we allow the designer to define a period of time after which the failed component comes back

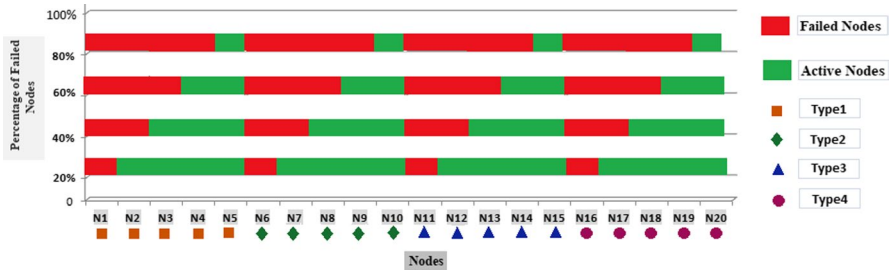

Fig. 7 Configuration of experiments 1 and 2

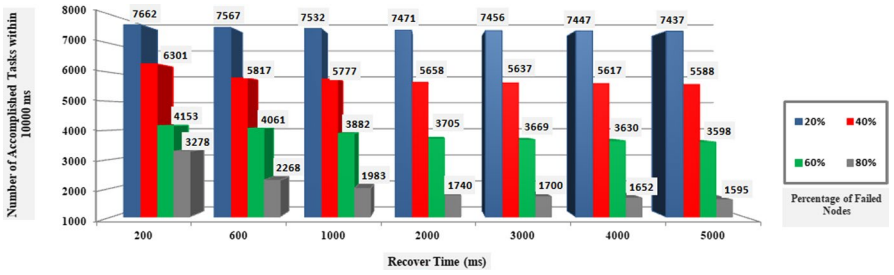

Fig. 8 Total accomplished tasks within 10^4 ms with respect to the percentage of failed nodes and the recover-time in the self-repair strategy

automatically to the active state, which we call the *recover-time*. Such a feature allows one to study different types of failures.

In the configuration of Fig. 7, we consider a system of 20 *Nodes*, 20 *Node-Monitors*, 40 *Devices* and 40 *Device-Monitors*. Each *Node* has a queue of 500 tasks and is connected to two devices. Each node has a monitor that checks its status every 30 ms. We propose to compute the number of accomplished tasks by the system within 10^4 ms of real-time executions by varying the percentage of failed nodes, and the results are depicted in Fig. 8.

Experiment 2 In this experiment, we focus on the impact of the migration recovery strategy on the performance of the system, based on the same configuration as in *Experiment 1*, to which we precisely give the type of each Node (Fig. 7). Note that our model supports heterogeneity as it allows the designer to define for each component a particular type. In the implementation of the migration strategy of the *recover-function* detailed in Fig. 4, the migration of tasks is only authorized for nodes of the same type. In Fig. 9, we compute the total number of accomplished tasks using the migration strategy while varying the recovery time and the percentage of failed nodes. Note that, in our model, when a node fails, its queue of tasks is migrated and distributed fairly between active nodes of the same type (see Fig. 10a, b).

Now, we aim to study the overhead introduced by the two recovery-strategies implemented using the external $C++$ functions, namely the replication process and

the migration process. In Table 3, we provide the total execution time of both functions with respect to the total number of nodes and also the number of simultaneously failed nodes. Indeed, in our model, the recovery functions are implemented in the *update* connector ($\{\gamma_{ij}\}_{i,j=[1,N]}^{i\neq j}$). This connector collects the set of all simultaneously failed nodes from all connected nodes, executes the implemented function, and returns the corresponding updated data for all nodes. This connector represents in our model a distributed and memoryless controller (as we have already proved in [48]). The recovery function is therefore executed to treat the set of detected failed nodes at the same time.

Experiment 3 In our model, each monitor checks dynamically and periodically (within a Q interval), the status of its associated node in order to detect failures and then to perform recovery. If a monitor's associated node does not respond after a certain number of trials (Nbr_C), it declares it failed. In this experiment, we propose to compare the number of detected failures to the number of nodes that fail effectively, by varying the recover-time, the check period, Q, and Nbr_C. To that end, we consider the configuration described in Table 4. We assume that all nodes can fail with respect to a random distribution.

In Fig. 11, we compute the number of detected and real failures with an Nbr_C equal to 3. In Fig. 12, we increase the number Nbr_C. All measures are taken for an interval of 10^4ms of real-time executions. One can notice that the choice of value for both Q and Nbr_C has a direct impact on the efficiency of a node monitor. More experiments and simulation analysis are also possible using the automatically generated code of our formal model. Note that this generated code is proven to be *semantically equivalent* to the initial model, which means that any properties already checked on the model still hold on its generated code.

Experiment 4 In this experimentation, we focus on a set of fault metrics that are strongly related to the safety validation of different autonomous systems applications. For example, in vehicle systems with an automotive safety integrity level, the effectiveness of implemented safety measures depends considerably on the values of these metrics [49]. One of the important metrics of the safety measure is the fault handling time interval (FHTI). If such a metric is greater than the fault tolerant time interval (FTTI), the safety measure must not be considered effective, and therefore, it cannot be taken into account for the safety validation. The FHTI contains the fault detection time interval (FDTI) and the fault reaction time interval (FRTI). The FDTI is the time taken by the system to detect the failure, and it corresponds in our system to the interval of time between the failure occurrence and the detection of the failure by the corresponding monitor. The FRTI corresponds in our model to the interval of time between the transitions *fail* and *update* of the node monitor. As given in Table 5, we have measured these failure metrics for different configurations with a total of 150 components. One can observe that these intervals can be influenced by different parameters of the model, such as the monitoring frequency Nbr_C or the monitoring period Q for the FDTI interval, and the number of simultaneously failed nodes for the FRTI interval.

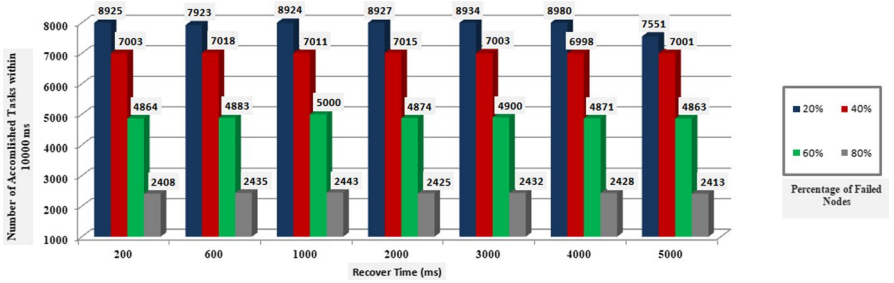

Fig. 9 The number of accomplished tasks within 10,000 ms by varying the percentage of failed nodes and the recover-time using the migration strategy

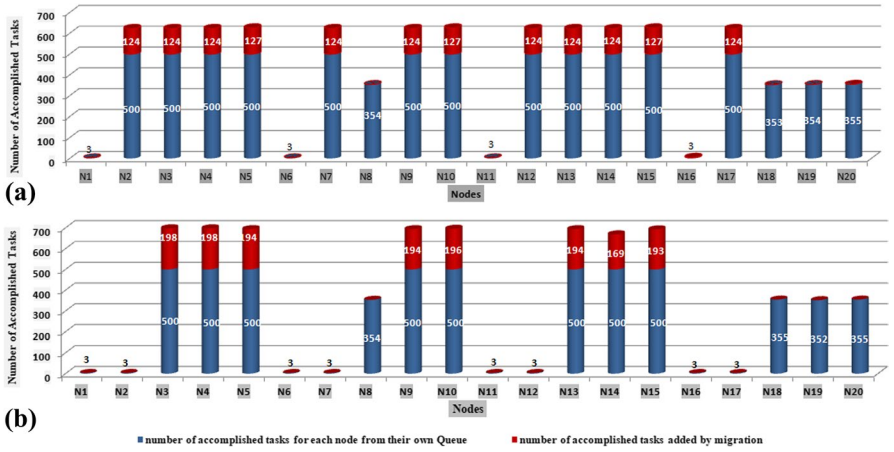

Fig. 10 The number of accomplished tasks for each Node within 10^4 ms and 200 ms of recover-time using the migration strategy: **a** 20% of failed nodes **b** 40% of failed

6 A case study: the DALA robot

In this section, we propose to apply our approach to a concrete autonomous system, namely the DALA Robot [16]. In particular, we focus on failure detection and recovery in the navigation module at the functional level of the robot.

6.1 DALA robot architecture

In general, robotic architectures are organized into several levels, which correspond to various temporal requirements (e.g., TREX [16]) or various levels of abstraction of functionality, such as the LAAS architecture [50]. The functional level, of most complex systems and robotic architectures, includes all the basic, built-in action and perception capabilities. These processing functions and control loops (e.g., image

Table 3 Execution time of the replication process and migration process

Nodes	40		60			80			100			150			
Failed nodes	5	10	5	10	20	5	10	20	5	10	20	5	10	20	30
Replication process (μs)	14	37	23	68	180	32	108	272	39	115	332	61	169	661	1045
Migration process (μs)	37	76	72	156	287	128	296	452	205	335	813	444	1064	1781	2193

Table 4 Configurations for experiment 3

Node	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}
Recover-time	20	20	40	40	60	60	80	80	100	100
Monitor	MN_1	MN_2	MN_3	MN_4	MN_5	MN_6	MN_7	MN_8	MN_9	MN_{10}
Q	5	20	20	10	30	10	40	10	20	20
Nbr_C	4	2	2	3	2	3	2	3	2	3
Node	N_{11}	N_{12}	N_{13}	N_{14}	N_{15}	N_{16}	N_{17}	N_{18}	N_{19}	N_{20}
Recover-time	120	120	140	140	160	160	180	180	200	200
Monitor	MN_{11}	MN_{12}	MN_{13}	MN_{14}	MN_{15}	MN_{16}	MN_{17}	MN_{18}	MN_{19}	MN_{20}
Q	60	30	70	40	80	50	90	50	100	50
Nbr_C	2	3	2	4	2	4	2	4	2	5

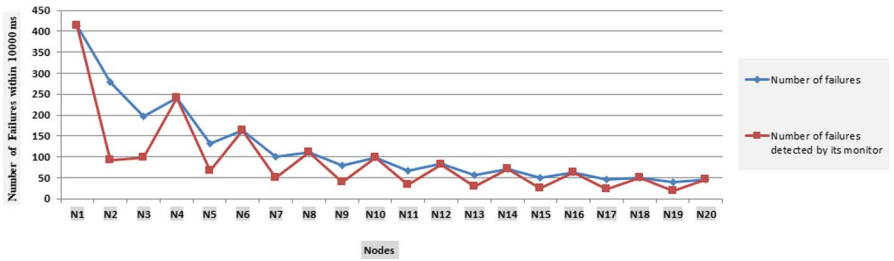

Fig. 11 Number of detected failures versus the number of concrete failures with $Nbr_C = 3$

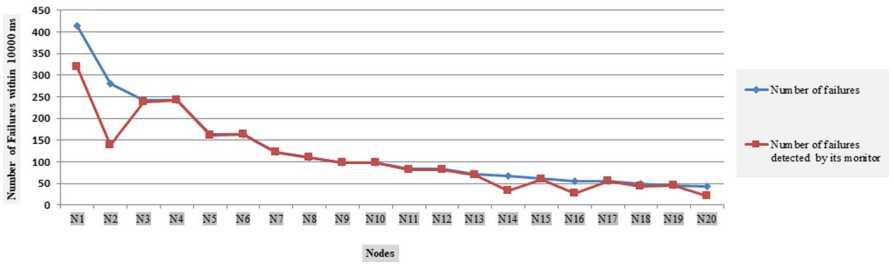

Fig. 12 Number of detected failures versus the number of concrete failures when varying Nbr_C

Table 5 Failure metrics measured for different configurations

$(Nbr_C), Q(ms)$	Config1 (1), 10^{-2}	Config2 (1), 2×10^{-2}	Config3 (1), 5×10^{-2}	Config4 (2), 8×10^{-2}	Config5 (2), 10^{-1}	Config6 (1), 2×10^{-1}	Config7 (2), 5×10^{-1}
FDTI (ms)	0.0157	0.0258	0.0711	0.1901	0.2459	0.2812	1.2102
FRTI (ms)	0.0101	0.0121	0.0132	0.0102	0.0142	0.0136	0.0127
FHTI (ms)	0.0258	0.0379	0.0843	0.2003	0.2601	0.2948	1.2229

processing, obstacle avoidance, and motion control) are encapsulated into controllable, communicating modules (see Fig. 13).

At LAAS, they have used GenoM [50] to develop these modules. Each module in the functional level of the LAAS architecture is responsible for a particular functionality of the robot. Complex modalities (such as navigation) are obtained by making modules work together. This architecture is used on all robots at LAAS (e.g., DALA, an iRobot ATRV; HRP2; Rackham, an iRobot B21; Jido, etc.). In this study, we focus on the functional level of DALA, which mainly includes two navigation modes, namely flat terrain navigation mode and rough terrain navigation mode. Each GenoM module of the functional level of DALA is built as a set of a hierarchy of components, as illustrated in Fig. 14. Each *Module* is composed of a set of services, a set of execution tasks, and a set of *Posters*. Each *Service* is a compound component which consists of a *Service controller* and *Activity*. Each *Execution Task* is a compound component which contains a *Timer* and a *Scheduler Activity*. In Fig. 14, we provide details about the mapping of each GenoM component to its corresponding modeling specification in our architecture. Using this analogy, our approach can be applied to any module at the functional level of DALA.

In this work, we focus on modules of the flat terrain navigation mode and we propose to analyze the NDD module of this navigation mode. The *NDD* module is one of the modules ensuring the Flat terrain navigation of the DALA robot. After proper initialization, it periodically (every 100 ms) recovers the current position (in the RFLEX module) (pos) poster and the obstacles (in the Aspect module) (Obs) poster. It produces the poster (speed) which will be used by the RFLEX module, which manages the low level robot wheels controller in order to control the speed

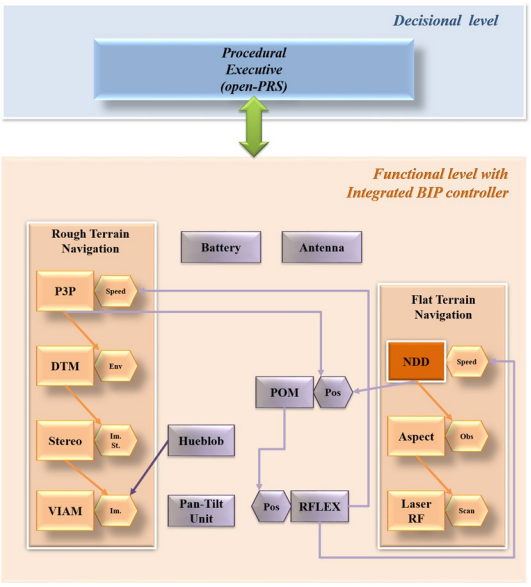

Fig. 13 DALA architecture

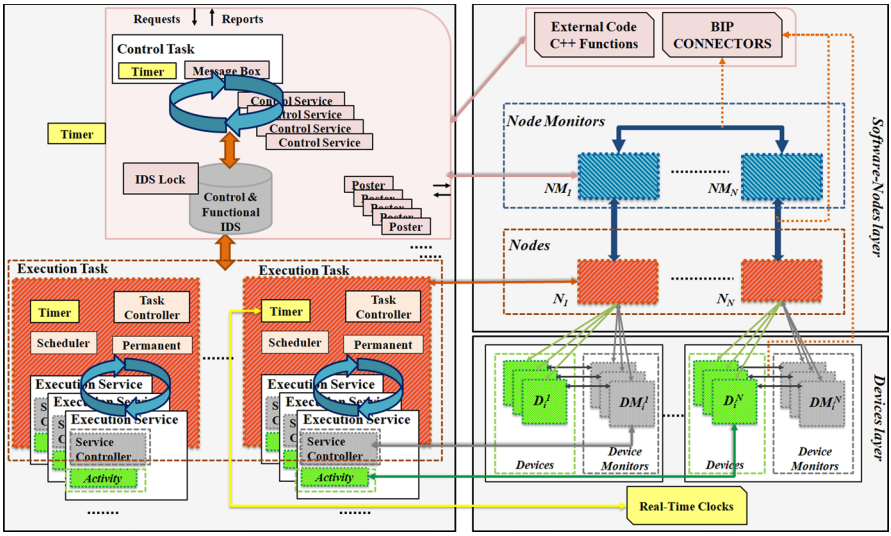

Fig. 14 A mapping of GenoM modules to our model components

of the robot. RFLEX also produces the current position of the robot based on odometry. This position is used by the POM module to generate the current position of the robot (see Fig. 13).

6.2 The NDD modeling and verification

The architecture of the NDD module is given in Fig. 15. It consists of five services, namely SetParams, SetDataSource, SetSpeed, Stop, and GoTo. Each service can be invoked by the higher (decisional) level according to the tasks that need to be achieved. Services can be execution services, which initiate activities that take time to execute, or control services, which take negligible time to execute and are responsible for setting and returning variable values. Three services, SetParams, SetDataSource, and SetSpeed, correspond to initializations of the navigation algorithm. The services Stop and GoTo are responsible for launching and stopping the path computation toward a given goal. Execution services are managed by execution tasks, which are responsible for launching and executing activities within the associated running services. Indeed, the NDD module contains five services and one execution task. Each service is modeled by composing the device and its monitor. The execution task is modeled by composing the node and its monitor. Figure 15 shows the obtained model of the NDD module with respect to the analogy given in Fig. 14.

Using the executable code of the NDD model generated by the real-time BIP, we have performed a set of experiments. In particular, we study the impact of the failure rate on the NDD module response time. In Figs. 16 and 17, we depict the results of several experiments related to the average response time of the NDD module for different failure rates. These failure rates are related to the failure of the *SetParams* service in Fig. 16 and also to the failure of the *SetSpeed* service

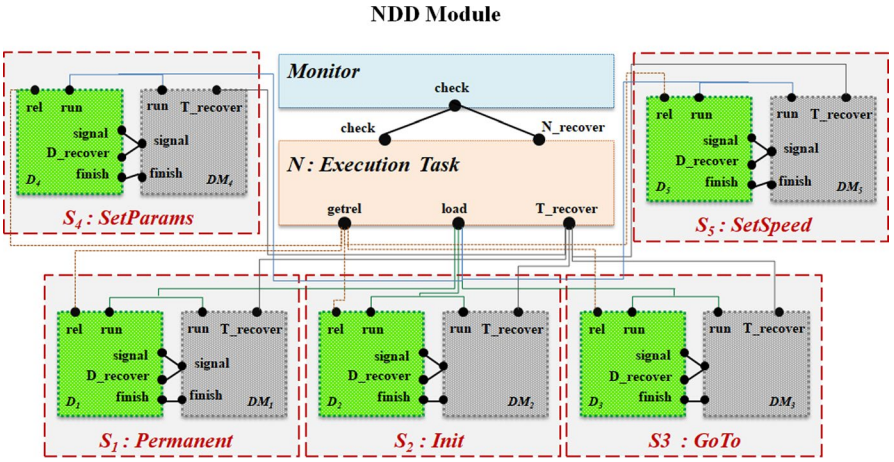

Fig. 15 NDD componentization prior to our approach

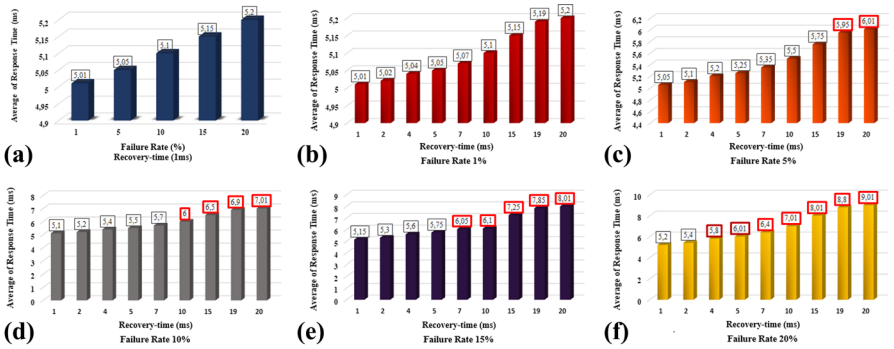

Fig. 16 The impact of failure rate and recovery-time of the *SetParams* service on the response time of the NDD module

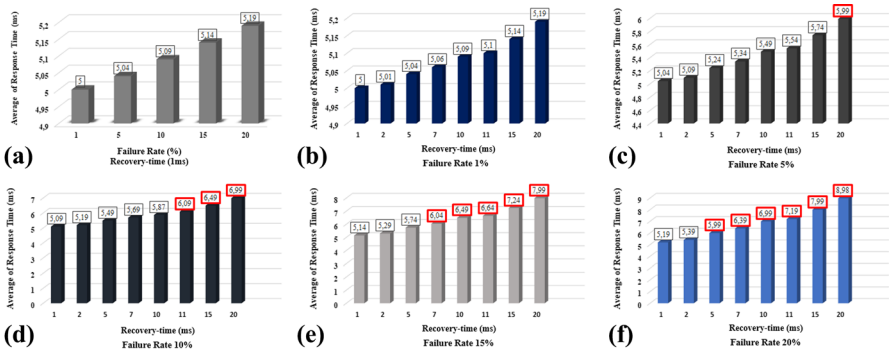

Fig. 17 The impact of failure rate and recovery-time of the *SetSpeed* service on the response time of the NDD module

in Fig. 17. The same set of experiments investigates the effect of self-recovery period variation on the overall response time of the NDD module. Indeed, recovery-time has an impact on the response time as well. Note that in the NDD module, the control task wakes up periodically every 100 ms, which means that the overall response time does not have to exceed this period of time. In Figs. 16 and 17, we highlight with red squares the cases where this timing constraint has been violated by one or more executions.

7 Conclusion

We presented a distributed and formal model for the specification, verification, and analysis of self-recovery behaviors in autonomous systems. Our approach is based on the combination of two frameworks to provide high level functional verification results and low-level performance analysis based on a stochastic version of the proposed model. We have focused on a completely distributed architecture with an implementation of different well-known failure-detection as well as recovery strategies. We have also applied our approach to a real-life case study corresponding to the navigation module of the DALA robot. As the key features of the overall approach, we highlight the following.

- Our model promotes a component-based design philosophy that allows for the detection component failures and supports the integration of recovery strategies.
- Our design flow is layered, which allows a separation between the application design and the lower-level system aspects, such as communication and computation.
- Both functional and non-functional system aspects are captured in our model, which enables a wide range of analysis using BIP and SBIP tools.

Possible research directions include the integration of strategies enabling failure prediction strategies. In such cases, failure is avoided by starting a recovery process before the failure is even detected. Defining reconfigurable models is also a possible research direction where the recovery strategy could be adapted on the fly with respect to the system status.

References

1. Oreizy P, Medvidovic N, Taylor RN (1998) Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering, IEEE, pp 177–186
2. Hölzl M, Rauschmayer A, Wirsing M (2008) Engineering of software-intensive systems: state of the art and research challenges. Software-Intensive Systems and New Computing Paradigms. Springer, New York, pp 1–44

3. Oquendo F (2016) Software architecture challenges and emerging research in software-intensive systems-of-systems. European Conference on Software Architecture. Springer, New York, pp 3–21
4. Gerostathopoulos I, Bures T, Hnetynka P, Keznikl J, Kit M, Plasil F, Plouzeau N (2016) Self-adaptation in software-intensive cyber-physical systems: from system goals to architecture configurations. J Syst Softw 122:378–397
5. Wang H, Zhong D, Zhao T (2019) Avionics system failure analysis and verification based on model checking. Eng Fail Anal 105:373–385
6. Pelliccione P, Tivoli M, Bucchiarone A, Polini A (2008) An architectural approach to the correct and automatic assembly of evolving component-based systems. J Syst Softw 81(12):2237–2251
7. Guarro S, Yau MK, Ozguner U, Aldemir T, Kurt A, Hejase M, Knudson M (2017) Formal framework and models for validation and verification of software-intensive aerospace systems. In: AIAA Information Systems-AIAA Infotech@ Aerospace, p 0418
8. Salvador R, Otero A, Mora J, de la Torre E, Sekanina L, Riesgo T (2011) Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems. In: Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs, IEEE, pp. 164–169
9. Pierce WH (2014) Failure-Tolerant Computer Design. Academic Press, New York
10. Stengel RF (1991) Intelligent failure-tolerant control. IEEE Control Syst Mag 11(4):14–23
11. Schneider M (1993) Self-stabilization. ACM Comput Surv (CSUR) 25(1):45–67
12. Kochte MA, Wunderlich H (2018) Self-test and diagnosis for self-aware systems. IEEE Design Test 35(5):7–18
13. Basu A, Bensalem S, Bozga M, Combaz J, Jaber M, Nguyen T, Sifakis J (2011) Rigorous component-based system design using the BIP framework. IEEE Softw 28(3):41–48
14. Nouri A, Mediouni BL, Bozga M, Combaz J, Bensalem S, Legay A (2018) Performance evaluation of stochastic real-time systems with the SBIP framework. IJCCBS 8(3/4):340–370
15. Nouri A, Bensalem S, Bozga M, Delahaye B, Jégourel C, Legay A (2015) Statistical model checking QoS properties of systems with SBIP. STTT 17(2):171–185
16. McGann C, Py F, Rajan K, Thomas H, Henthorn R, McEwen RS (2008) A deliberative architecture for AUV control. In: Proceedings of the 2008 IEEE International Conference on Robotics and Automation, ICRA, IEEE, pp 1049–1054
17. Psaier H, Dustdar S (2011) A survey on self-healing systems: approaches and systems. Computing 91(1):43–73
18. Pereira EG, Pereira R, Taleb-Bendiab A (2005) Performance evaluation for self-healing distributed services. In: Proceedings of the 11th International Conference on Parallel and Distributed Systems, ICPADS, pp 135–139
19. McMinn P (2004) Search-based software test data generation: a survey. Softw Test Verif Reliab 14(2):105–156
20. Briand L, Nejati S, Sabetzadeh M, Bianculli D (2016) Testing the untestable: model testing of complex software-intensive systems. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp 789–792
21. Deonandan I, Valerdi R, Lane JA, Macias F (2010) Cost and risk considerations for test and evaluation of unmanned and autonomous systems of systems. In: Proceedings of the 2010 5th International Conference on System of Systems Engineering, IEEE, pp 1–6
22. Krishna CM (2014) Fault-tolerant scheduling in homogeneous real-time systems. ACM Comput Surv (CSUR) 46(4):1–34
23. Devaraj R, Sarkar A, Biswas S (2017) Fault-tolerant preemptive aperiodic RT scheduling by supervisory control of TDES on multiprocessors. ACM Trans Embed Comput Syst (TECS) 16(3):1–25
24. Devaraj R, Sarkar A Resource-optimal fault-tolerant scheduler design for task graphs using supervisory control. IEEE Trans Ind Inform
25. Ye L, Lin LZ (2010) Study of superconducting fault current limiters for system integration of wind farms. IEEE Trans Appl Supercond 20(3):1233–1237
26. Azad SP, Niazmand B, Janson K, George N, Oyeniran AS, Putkaradze T, Kaur A, Raik J, Jervan G, Ubar R (2017) From online fault detection to fault management in network-on-chips: a ground-up approach. In: IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS). IEEE 2017, pp 48–53

27. Hu J, Bhowmick P, Jang I, Arvin F, Lanzon A A decentralized cluster formation containment framework for multirobot systems. IEEE Trans Robot
28. Filippidis I, Dimarogonas DV, Kyriakopoulos KJ (2012) Decentralized multi-agent control from local LTL specifications. In: Proceedings of the 2012 IEEE 51st IEEE Conference on Decision and Control (CDC), IEEE, pp 6235–6240
29. Weyns D, Iftikhar MU, de la Iglesia DG, Ahmad T (2012) A survey of formal methods in self-adaptive systems. In: Fifth International C* Conference on Computer Science and Software Engineering, C3S2E '12, pp 67–79
30. Iftikhar MU, Weyns D (2012) A case study on formal verification of self-adaptive behaviors in a decentralized system. In: Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA, pp 45–62
31. Güdemann M, Ortmeier F, Reif W (2006) Safety and dependability analysis of self-adaptive systems. In: Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006), IEEE, pp 177–184
32. Mian NA, Ahmad F (2018) Agent based architecture for modeling and analysis of self adaptive systems using formal methods. Int J Adv Comput Sci Appl 9(1):563–567
33. Salehie M, Tahvildari L (2009) Self-adaptive software: landscape and research challenges. ACM Trans Auton Adapt Syst (TAAS) 4(2):1–42
34. Dashofy EM, Van der Hoek A, Taylor RN (2002) Towards architecture-based self-healing systems. In: Proceedings of the First Workshop on Self-Healing Systems, pp 21–26
35. Garlan D, Schmerl B (2002) Model-based adaptation for self-healing systems. In: Proceedings of the First Workshop on Self-Healing Systems, pp 27–32
36. Oreizy P, Gorlick MM, Taylor RN, Heimhigner D, Johnson G, Medvidovic N, Quilici A, Rosenblum DS, Wolf AL (1999) An architecture-based approach to self-adaptive software. IEEE Intell Syst Appl 14(3):54–62
37. Putze F, Ihrig T, Schultz T, Stuerzlinger W (2020) Platform for studying self-repairing auto-corrections in mobile text entry based on brain activity, gaze, and context. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pp 1–13
38. Oquendo F (2016) Formally describing the architectural behavior of software-intensive systems-of-systems with sosadl. In: Proceedings of the 21st International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, pp 13–22
39. Ben-Rayana S, Bozga M, Bensalem S, Combaz J (2016) Rtd-finder: A tool for compositional verification of real-time component-based systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, pp 394–406
40. Gurunathan A, Viswanatham VM (2017) Autonomic performance enhancement environment for websphere application server. Int J Pure Appl Math 116(23):719–731
41. Simmons R, Pecheur C, Srinivasan G (2000) Towards automatic verification of autonomous systems. In: Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113), Vol. 2, pp 1410–1415
42. Ehrig H, Ermel C, Runge O, Bucchiarone A, Pelliccione P (2010) Formal analysis and verification of self-healing systems. In: International Conference on Fundamental Approaches to Software Engineering, Springer, pp 139–153
43. Basu A, Bozga M, Sifakis J (2006) Modeling heterogeneous real-time components in bip. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), IEEE, pp 3–12
44. Mediouni BL, Nouri A, Bozga M, Dellabani M, Legay A, Bensalem S (2018) S BIP 2.0: Statistical model checking stochastic real-time systems. In: International Symposium on Automated Technology for Verification and Analysis, Springer, pp 536–542
45. Bliudze S, Sifakis J (2008) The algebra of connectors: structuring interaction in BIP. IEEE Trans Comput 57(10):1315–1330
46. Park T, Byun I, Kim H, Yeom HY (2002) The performance of checkpointing and replication schemes for fault tolerant mobile agent systems. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems, 2002. IEEE, pp 256–261
47. Glass M, Lukasiewycz M, Streichert T, Haubelt C, Teich J (2007) Reliability-aware system synthesis, design. Automation Test in Europe Conference Exhibition pp 1–6
48. Ben-Hafaiedh I, Graf S, Quinton S (2011) Building distributed controllers for systems with priorities. J Log Algeb Prog 80(3–5):194–218

49. Köhler A, Bertsche B (2021) Cyclisation of safety diagnoses: influence on the evaluation of fault metrics. In: Annual Reliability and Maintainability Symposium (RAMS). IEEE pp 1–7
50. Fleury S, Herrb M, Chatila R (1997) G^{en}om: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In: Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS, IEEE, 1997, pp 842–849

Authors and Affiliations

Imene Ben Hafaiedh[1] · **Maroua Ben Slimane**[2]

[1] LIPSIC Laboratory, Higher Institute of Computer Science (ISI), University of Tunis El Manar (UTM), Tunis, Tunisia

[2] LIPSIC Laboratory, University of Tunis El Manar (UTM), Tunis, Tunisia