Contemporary Mathematics
https://ojs.wiserpub.com/index.php/CM/

Research Article

Ensuring Reliability in Self-Adaptive Systems: A Framework for Formal Specification and Verification

Muhammad Abdul Basit Ur Rahim

Department of Computer Engineering and Computer Science, California State University, Long Beach, USA
E-mail: m.basit@csulb.edu

Received: 9 December 2024; **Revised:** 15 January 2025; **Accepted:** 24 February 2025

Abstract: Self-adaptive systems are designed to maintain functionality and tolerate faults in dynamic and unpredictable environments by adjusting their behavior during runtime. Ensuring the correctness of these adaptations is crucial, especially in safety-sensitive applications. This paper presents a formal verification methodology to validate such systems' reliability and adaptive correctness. The approach is demonstrated through a mining robotic arm case study, where system behaviors are formally specified using Duration Calculus. These specifications are then systematically mapped to a PRISM model. Using the PRISM model checker, the design is rigorously verified against both functional and non-functional constraints, which include liveness, safety, and deadlock-freeness. The results indicate that the system maintains correct operation with over 99.9% reliability, even under conditions of uncertainty. This framework enhances the target system's fault tolerance and broad applicability to other self-adaptive systems operating in dynamic environments.

***Keywords*:** self-adaptive systems, model-checking, formal specification, formal verification, reconfiguration

MSC: 65L05, 34K06, 34K28

Abbreviation

DC	Duration Calculus
FSA	Finite State Automata
PTA	Probabilistic Timed Automata
CTL	Computational Tree Logic
PCTL	Probabilistic Computational Tree Logic

1. Introduction

A self-adaptive system can manage itself to achieve its high-level objectives [1] and reconfigure itself to adapt to runtime changes or errors [2]. It must ensure the correctness of performing its functions, and when an emergency occurs, it can reconfigure itself correctly [3]. However, due to growing complexity and dynamism, designing a self-adaptive system

DOI: https://doi.org/10.37256/cm.6520256223

takes time and effort. Many variables are unknown during the design phase, but can only be available at runtime. Another challenge is that there always exist many sources of runtime uncertainties [4]. These challenges need to be addressed in the early stage of the design process so that the design flaws can be identified.

Robotics is essential to performing simple and repetitive work; they are even indispensable and has significantly contributed to dangerous work if performed by humans. One example of indispensable robotics usage is in the mining industry. Robots lift heavy loads, work in hazardous environments, or deal with toxic substances in mining. Robots have assisted companies in preventing accidents, increasing productivity, and reducing costs. However, robots can hurt themselves in some situations. Integrating safety into the robot systems and verifying the correctness and integrity of internal logic is essential to protect itself from damage during an emergency. Otherwise, design errors can lead to failure or even catastrophic consequences. In this context, many prior works have addressed that the most appropriate technique of assuring its runtime correctness is through formal methods at an early design stage [5].

Limitations of existing verification methodologies. Self-adaptive system verification methods currently in use have significant drawbacks, especially in intricate robotic applications. The difficulties caused by non-deterministic transitions that are a part of runtime uncertainties and the incapacity to fully control state explosion during formal verification are two examples. For example, current tools frequently have trouble modeling and validating dynamic reconfigurations and probabilistic behaviors in distributed architectures, making systems susceptible to design flaws that show up while they are in use. Developing strong and dependable self-adaptive systems is hampered by these gaps, underscoring the need for sophisticated approaches that successfully handle these issues while guaranteeing safety, accuracy, and adaptability in practical situations.

Motivation. Formal methods play a crucial role during the design phase of self-adaptive systems by narrowing the uncertainty gap between specification and implementation [5]. These methods, typically based on mathematical models or formal logic, enable the specification of both functional and non-functional requirements, which can then be simulated, modeled, and verified as if operating in real-world conditions [6]. This approach is particularly beneficial for addressing the intrinsic dynamism of system components at runtime and identifying design flaws before development. Formal verification becomes especially critical in the early stages of the design process, where field tests or simulations may be infeasible. It ensures that self-adaptive systems meet their objectives, providing safety and reliability in hazardous environments where direct human interaction is limited.

In this work, we use a mining robotic arm system as a case study and formally specify, model, and verify its self-adaptivity and reconfigurability. The proposed methodology ensures the correctness of the design and operational integrity.

The outline of the paper is as follows. Section 2 presents the related work. Section 3 describes the preliminaries. Section 4 presents a brief overview of our research process. Section 5 formally specifies the functional and self-adaptive requirements of the mining robotic arm system using DC implementables. Section 6 introduces a method for directly mapping DC implementables to a PRISM model. In Section 7, the results are discussed. Section 8 concludes the research.

2. Related work

Self-adaptive systems incorporate internal components such as feedback mechanisms, analyzers, and decision-makers to manage adaptations and reconfigurations autonomously. One of the main challenges that arises in these systems is dealing with uncertainties during runtime. Therefore, effective strategies for managing these uncertainties and ensuring the correctness of adaptations are essential. In this section, we first discuss some of the challenges faced by self-adaptive systems, as highlighted in existing literature. Then, we review the common formal methods used to specify self-adaptive systems. Finally, we examine verification methods that can ensure and guarantee correctness, considering defined properties and constraints.

2.1 *Challenges of self-adaptive systems*

A key component of self-adaptive systems is their ability to handle changes during runtime, which sets them apart from other systems. Self-adaptive systems must account for both offline activities during the design phase and online activities during runtime. Therefore, completing the design phase is only half of the overall work [4]. During runtime, these systems encounter various uncertainties stemming from the external environment or from their internal components. Consequently, self-adaptive systems need to design and implement adaptation logic that effectively addresses the uncertainties inherent in online activities [7]. This adaptation must be carried out appropriately and robustly, enabling the systems to adjust to changing conditions, partial failures, and errors during operations.

A key question in designing a self-adaptive system is how to ensure the correctness of its self-adaptability [5]. Many existing studies suggest that providing formal specifications and verification during the early design phase is a suitable approach to achieving this important goal. However, among the research on autonomous robotic systems, only about one-third of the studies utilize formal specification, modeling, or validation tools to verify correctness [6]. According to Luckcuck et al. [6], the limited use of formal methods is primarily due to a lack of available methods and tools for conducting such formal analyses. Often, models or specifications of similar components are incompatible and tied to specific tools, which are only suitable for certain modeled components or properties of interest. In light of this shortage of formal tools, applying formal methods remains a valuable strategy for addressing these challenges [8].

2.2 *Formal verification of self-adaptive system*

Javier et al. [6] reviewed hundreds of papers that utilize various methods for specifying and verifying autonomous robotic systems that require self-adaptivity. We will highlight the most commonly used methods based on their findings.

According to their survey, formalism is typically employed to specify a system's behaviors and properties. When it comes to behavior specification, set-based formalisms are the most prevalent approach. For instance, Weyns et al. [9] employ Z-model approaches, Liang et al. [10] use a Java animator for Z specifications, and Tarasyuk et al. [11] utilize Event-B specifications to describe the system's behaviors. Set-based methods utilize set-theoretic representations and data manipulation. Their main advantage lies in their ability to capture the data structures of the described system accurately. However, these methods are limited in their capacity to represent the full spectrum of system behaviors, particularly uncertain behaviors that may occur during runtime. The studies mentioned above combine set-based methods with models implemented and verified in PRISM, allowing for system refinement and probabilistic assessment of its reliability and performance.

An alternative approach involves using State-Transition Formalisms, such as Petri Nets and Finite-State Automata (FSA), to specify the behaviors of a state-transition system. These formalisms are effective in capturing time and probabilistic transitions. Some researchers integrate temporal logic with FSA [12], while others utilize Petri Nets to represent the abstract architecture of robotic agents [13–15]. Additionally, Iftikhar et al. [16] employ the UPPAAL model checker with timed automata to create input models and verify them against temporal logic properties. Furthermore, the works cited in [17, 18] extend FSA to model the physical environment and correspond to the robot's actions. These methods specify the discrete events of a system, making them particularly useful for detailing behaviors during the design phase.

2.3 *Verification using model checking*

Regarding formal verification tools, model checking is the most popular method than Program Model Checkers, Theorem Provers, and others [6]. According to Konur et al. [19], PRISM can be used to represent quantities such as "the probability a robot eventually reaches the nest", "the probability that the energy in the system is greater than E", etc., as well as standard temporal properties. PRISM can also compute the minimum or maximum probability over specific configurations or parameters of a model, producing a form of best or worst-case analysis.

Some works use selected formal specification methods, which are then modeled and verified in the PRISM model checker. Celaya et al. [20] uses property-driven design and PCTL directly from PRISM's syntax to solve the DTMC

problem. Konur et al. use PCTL syntax and PRISM for model checking [21, 22]. Massink et al. [23] use the Bio-PEPA tool and PRISM model checking. According to our review, the PRISM model checker can be a suitable tool for verifying the probabilistic properties of self-adaptive systems.

3. Preliminaries

Model Checking. It is a formal verification technique for analyzing the system's behaviors and verifying finite state systems [24] against all possible behaviors of the system in question. The system is modeled as a state transition diagram and constraints are specified using temporal logic. An efficient search procedure is used to determine whether or not the state transition model satisfies the constraints.

Temporal logic is any system of rules and symbolism for representing and reasoning about propositions qualified in terms of time (Bengtsson 1995). For specification, PRISM supports Probabilistic Computation Tree Logic (PCTL), an extension of Computational Tree Logic (CTL).

Duration Calculus. It is an interval logic that describes the real-time behavior of dynamic systems [25]. It is used to specify an interval temporal logic and specify the functionality of the real-time system at the abstraction, concrete and low levels. The DC expressively specifies the embedded and real-time systems. Three essential dimensions (reasoning about data, communication, and real-time aspects) are required for the verification of embedded hardware and software systems. DC can express all these dimensions [26], which are required for model self-adaptive system. Numerous prominent approaches are available that perform verification of temporal specifications by translating the problem into an automata-theoretic setting [25, 26].

DC has a subset known as DC-implementable, which we have used to describe the detailed model for real-time systems [27, 28]. It can also describe the scenarios for changing the values of parameters. The most important reason for using DC is that the DC constraints are mathematically provable.

Table 1 presents some symbols for the specification of duration calculus. The ceilings $\lceil\rceil$ are used to specify the state. The right arrow is used for transition. The symbol ε with a right arrow indicates the transition with delay. The right arrow with zero indicates the control is initially in that state. DC implementables describe the transition among the states and set the invariants to stabilize the system.

We preferred using Duration Calculus (DC) over Probabilistic Timed Automata (PTA) for the following reasons:

Expressiveness: DC is a more expressive formalism that can capture data, communication, and real-time aspects required for modeling self-adaptive systems. PTA does not provide the same level of expressiveness needed to model complex self-adaptive behavior.

Mathematical Provability: as DC constraints are mathematically provable, which is an advantage over PTA for ensuring the correctness of specifications through formal mathematical reasoning.

Table 1. Duration calculus implementables

Symbol	Description
$\lceil x \rceil$	region
$\longrightarrow$	followed-by
$\longrightarrow_0$	followed-by-initially
$\stackrel{\varepsilon}{\rightarrow}$	delay transition
$\stackrel{\leq\varepsilon}{\longrightarrow}$	upto
$\stackrel{\varepsilon}{\rightarrow}_0$	followed-by-initially
;	chop

4. Verification methodology

Figure 1 demonstrates the verification process. Based on functional requirements, the system is decomposed into small modules and formalized the functional requirements. The case study is formally specified using DC implementables. The formal requirements are further mapped to the input language of the PRISM model-checking tool. Later, the PRISM model-checker simulates and verifies the system's correctness, safety, liveness, fairness, deadlock-freeness, and reconfigurability. The constraints are specified using the PCTL - a query language consisting of the path and state formulas. The model-checker executes the formal model and verifies it against the functional and non-functional constraints, resulting in satisfied or unsatisfied. If it is unsatisfied, correct the model and reiterate the process.

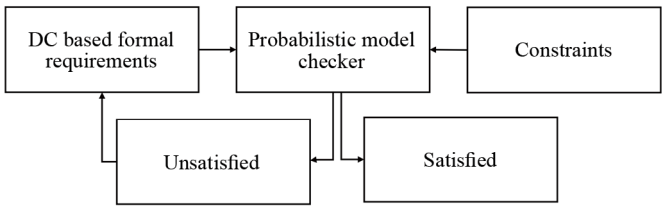

Figure 1. Verification process

The proposed methodology enhances safety and reliability by ensuring correctness under uncertainty through formal modeling and probabilistic verification, drawing on real-world experiences with formally modeling self-adaptive systems. As the mining robotic arm case study illustrates, this method successfully addresses issues like runtime uncertainties and design flaws, providing a scalable framework for enhancing operational efficiency and averting failures in dynamic environments.

5. Robotic Arm - A case study

We have modeled the mechatronic arm, which is a mining robot. We have formally specified the system's requirements in Duration Calculus, followed by a proposed mapping technique to automate the conversion from DC implementables to a model in PRISM, a probabilistic model checker tool to perform the probabilistic analysis.

The robotic arm system comprises an arm and a bucket. Two separate embedded micro-controllers control the arm and the bucket. The arm sub-system controls how far the arm should reach to safely complete the task without harming the robot. The bucket sub-system controls the open or closed state of the bucket in operation. Figure 2 illustrates the complete Robotic Arm system overview.

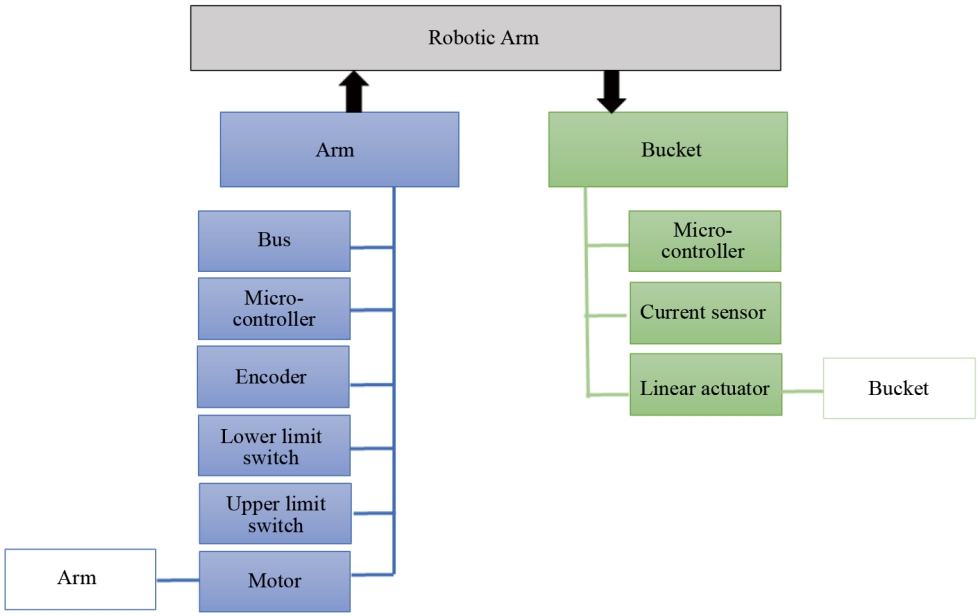

Figure 2. Robotic Arm system

5.1 *The Arm system*

Figure 3 displays the state machine diagram for the Arm system with six components: Bus, Controller, Lower Limit Switch, Upper Limit Switch, Encoder, and Motor. The Bus initializes target positions, the Controller orchestrates actions, and the Encoder tracks Motor position. Limit Switches ensure system safety. The system transitions through states based on signals, responding to events in the dynamic working environment.

The Arm system initiates with the Controller in a setup stage, where the Bus is in bus_ready state, the Motor is set to motor_stop state with a random position within a specified range, and the Encoder starts at a lower threshold (lower_thresh). After setup, the Controller enters idle, ready for signals. If an issue arises, it transitions to a halt state for system protection. Upon receiving a new target from the Bus, the Controller, in the bus_waiting state, calculates the difference between the current position and the target. It then adjusts the Motor's position incrementally, with the Encoder tracking in real-time. This process continues until the target is reached, triggering a bus_arrive signal to the Bus. The Controller returns to idle, and the Bus resets to bus_ready. This process applies to both increment and decrement scenarios.

The Controller updates the two Limit Switches with the encoder_recorded_position, reflecting the Motor's real-time position. When the Motor's position surpasses the normal operational range ($\leq$lower_thresh or $\geq$upper_thresh), the Lower_Switch_Limit sends L_switch_pushed (same to the U_switch_pushed) signal to the Controller, which moves to the state of lower_limit or upper_limit. The Controller receives such a signal; it moves to either lower_limit or upper_limit. After that, it moves to halt ($<$lower_thresh or $>$upper_thresh) or idle (equals to lower_thresh or upper_thresh).

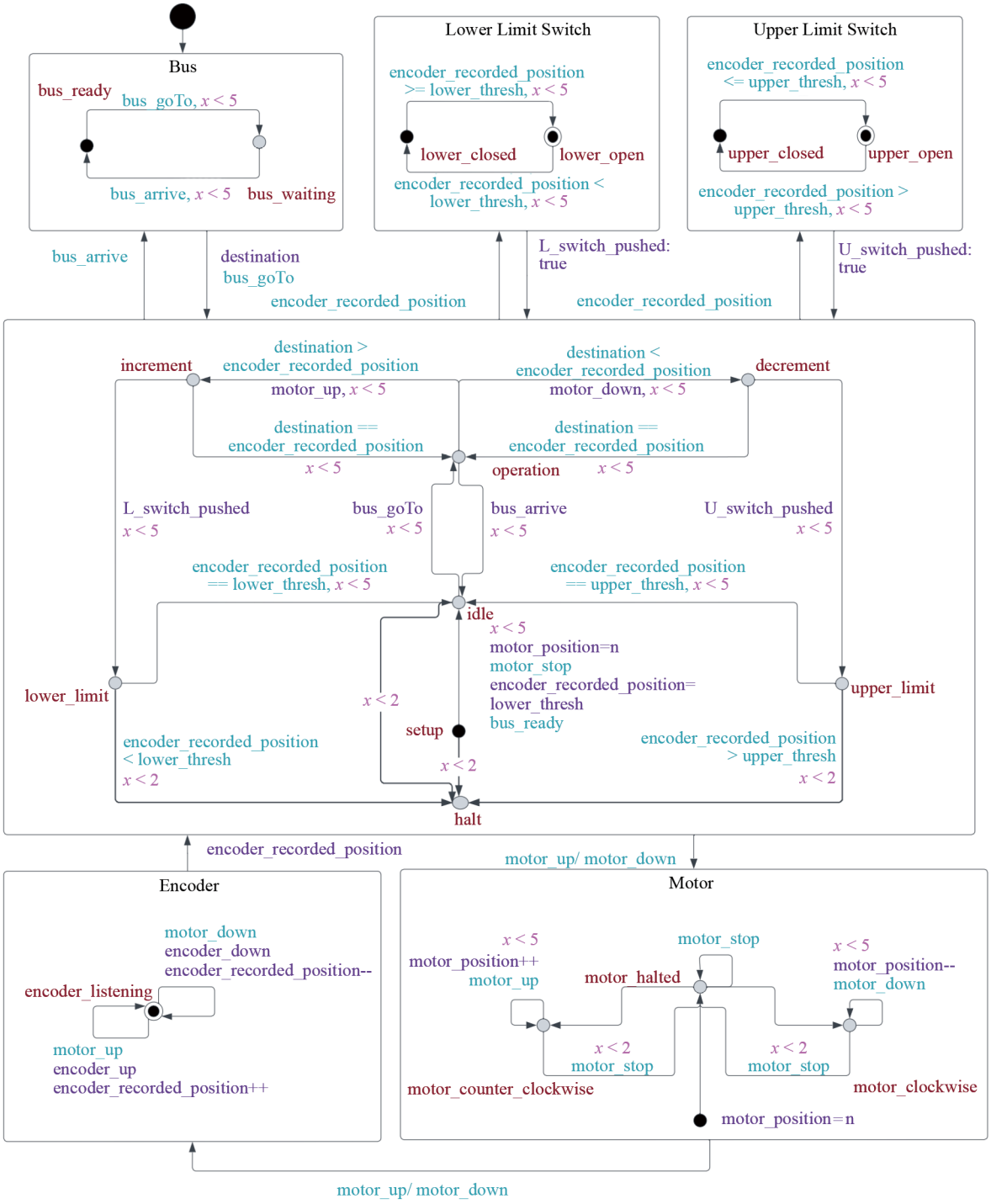

Figure 3. State machine diagram for Arm

Formal Specification of Arm. Table 2 provides a detailed Duration Calculus (DC) based formal specification for the Arm system depicted in Figure 3. The Controller, as the main component, initializes in the setup phase (Init-1). The Encoder's position is set to lower_thresh (Init-2), Limit Switches are open (Init-3), Motor's position is randomly set within the operational range (Init-4), and Bus is in the bus_ready state (Init-5), with the Motor in motor_stop (Init-6). After setup,

the system transitions to idle (Seq-1). In case of setup issues, such as the Encoder's position exceeding upper_thresh or falling below lower_thresh, the system enters a halt state (Stab-1).

Table 2. DC implementables specifying the Arm

Label	Transactions
Init-1:	$\lceil\rceil \vee \lceil setup \rceil; true,$
Init-2:	$\lceil\rceil \vee encoder\_recorded\_position = lower\_thresh; true,$
Init-3:	$\lceil\rceil \vee (\lceil lower\_open \rceil; true \cdot \lceil upper\_open \rceil; true),$
Init-4:	$\lceil\rceil \vee motor\_position = n,\ n \in (lower\_thresh, upper\_thresh); true\ ,$
Init-5:	$\lceil\rceil \vee \lceil bus\_ready \rceil; true,$
Init-6:	$\lceil\rceil \vee \lceil motor\_stop \rceil; true,$
Seq-1:	$\lceil setup \rceil \rightarrow \lceil idle \rceil,$
Seq-2:	$\lceil motor\_stop \rceil \rightarrow \lceil bus\_arrive \rceil,$
Seq-3:	$\lceil bus\_arrive \rceil \rightarrow \lceil bus\_ready \rceil,$
Seq-4:	$\lceil bus\_arrive \rceil \rightarrow \lceil idle \rceil,$
Stab-1:	$\lceil setup \wedge (encoder\_recorded\_position < lower\_thresh \vee$ $encoder\_recorded\_position > upper\_thresh) \rceil \rightarrow \lceil halt \rceil,$
Stab-2:	$\lceil idle \wedge bus\_goTo \rceil \rightarrow \lceil operation \rceil,$
Stab-3:	$\lceil operation \wedge (destination < encoder\_recorded\_position) \rceil \rightarrow \lceil decrement \rceil,$
Prog-1:	$\lceil decrement \rceil \xrightarrow{<5} \lceil lower\_limit \rceil,$
Stab-4:	$\lceil operation \wedge encoder\_recored\_position < destination \rceil \rightarrow \lceil increment \rceil,$
Prog-2:	$\lceil increment \rceil \xrightarrow{<5} \lceil upper\_limit \rceil,$
Stab-5:	$\lceil decrement \wedge encoder\_recored\_position > destination \rceil \rightarrow \lceil decrement \rceil,$
Stab-6:	$\lceil upper\_open \wedge encoder\_recorded\_position > upper\_thresh \rceil \rightarrow \lceil U\_switch\_pushed \rceil,$
Stab-7:	$\lceil lower\_open \wedge encoder\_recorded\_position < lower\_thresh \rceil \rightarrow \lceil L\_switch\_pushed \rceil,$
Stab-8:	$\lceil motor\_clockwise \wedge motor\_down \rceil \rightarrow \lceil update\ motor\_position \rceil,$
Stab-9:	$\lceil motor\_counter\_clockwise \wedge motor\_up \rceil \rightarrow \lceil update\ motor\_position \rceil,$
Stab-10:	$\lceil encoder\_listening \wedge motor\_down \rceil \rightarrow \lceil update\ encoder\_recorded\_position \rceil,$
Stab-11:	$\lceil encoder\_listening \wedge motor\_up \rceil \rightarrow \lceil update\ encoder\_recorded\_position \rceil,$
Stab-12:	$\lceil (increment \vee decrement) \wedge (encoder\_recorded\_position == motor\_position) \rceil \rightarrow \lceil motor\_stop \rceil.$

In idle, if the Bus sends a bus_goTo signal with a destination, and the Encoder's position is within the safety range, the Controller compares the destination with the encoder_recorded_position (Stab-2). If the destination is higher, it transitions to increment (Stab-4); otherwise, it transitions to decrement (Stab-3). The Encoder continuously updates the Controller on the Motor's real-time position (Stab-5). In case of excessive, the Motor increment state triggers L_switch_pushed (Stab-7), and the Controller moves to lower_limit within 5 seconds (Prog-1). Similar logic applies to Stab-6 and Prog-2. The Motor_position (Stab-9) and encoder_recorded_position (Stab-10) update during motor spin. Once encoder_recorded_position matches the destination, the Controller signals motor_stop (Stab-12). In Seq-4, the Motor signals bus_arrive (Seq-2), transitioning Bus to bus_ready (Seq-3), and the Controller to idle (Seq-4).

5.2 *The bucket system*

The bucket system consists of a controller, linear actuators, and current sensors. The controller receives feedback from current sensors and controls the bucket through linear actuators, regulating motor speed and direction. The current sensors monitor Amperes flowing through the motors, allowing the controller to track and respond to variations. When the bucket is between fully opened or closed positions, its precise location is unknown, but the controller monitors the direction of the linear actuators. If the current surpasses a threshold, indicating high resistance, and persists, the controller takes corrective action to prevent motor burnout. The model aims to demonstrate the control algorithm's ability to handle incidents of excessive current safely and effectively.

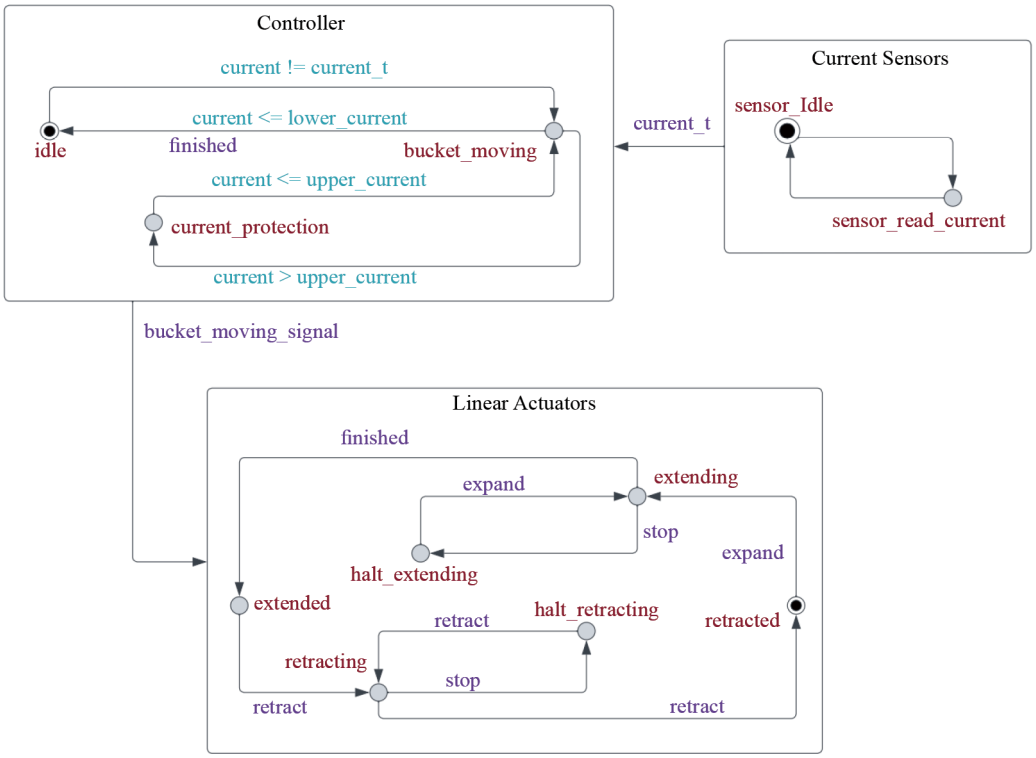

Figure 4. State machine diagram for bucket

Figure 4 illustrates the bucket system, focusing on the bucket controller's states and signals. The controller operates in three states: idle, bucket_moving, and current_protection. In idle, it reads current from sensors, updating the controller and initiating actions if current changes. If the current is below a threshold, it returns to idle; if above, it enters current_protection to prevent burnout. Linear actuators receive signals to expand or retract, transitioning between states. Once the process completes, the controller returns to idle, ready for the next signal.

Formal Specification of Bucket System. Table 3 provides a detailed DC-based formal specification for the Bucket system depicted in Figure 4. The system includes the bucket controller and linear actuators, with specific behaviors defined using DC implementables. Initially, the bucket controller is in the idle state, linear actuators are retracted, and the current in the controller is set as lower_current. The sensor updates the current_t to the controller. If the current values differ, the system transitions to bucket_moving, and depending on the current's range, it either remains in bucket_moving or returns to idle. In bucket_moving, the controller sends actuate signals to linear actuators, initiating iterative expand or retract steps. The actuators respond to stop or finished signals, transitioning between states until the controller returns to idle when the task is complete.

Table 3. Implementables specifying the bucket

Label	Transactions
Init-1:	$\lceil\rceil \vee \lceil idle \rceil; true,$
Init-2:	$\lceil\rceil \vee retracted; true,$
Init-3:	$\lceil\rceil \vee current = lower\_current; true,$
Init-4:	$\lceil\rceil \vee sensor\_idle; true,$
Stab-1:	$\lceil idle \wedge current \neq current\_t \rceil \rightarrow \lceil bucket\_moving \rceil,$
Stab-2:	$\lceil bucket\_moving \wedge current \leq lower\_current \rceil \rightarrow \lceil idle \rceil,$
Stab-3:	$\lceil bucket\_moving \wedge current \geq upper\_current \rceil \rightarrow \lceil current\_protection \rceil,$
Stab-4:	$\lceil bucket\_moving \wedge bucket\_moving\_signal == expand \rceil \rightarrow \lceil extending \rceil,$
Stab-5:	$\lceil bucket\_moving \wedge bucket\_moving\_signal == retract \rceil \rightarrow \lceil retracting \rceil,$
Stab-6:	$\lceil extending \wedge bucket\_moving\_signal == stop \rceil \rightarrow \lceil halt\_extending \rceil,$
Stab-7	$\lceil retracting \wedge bucket\_moving\_signal == stop \rceil \rightarrow \lceil halt\_retracting \rceil,$
Stab-8:	$\lceil halt\_extending \wedge bucket\_moving\_signal == expand \rceil \rightarrow \lceil extending \rceil,$
Stab-9:	$\lceil halt\_retracting \wedge bucket\_moving\_signal == retract \rceil \rightarrow \lceil retracting \rceil,$
Stab-10:	$\lceil expanding \wedge bucket\_moving\_signal == finished \rceil \rightarrow \lceil extended \rceil,$
Stab-11:	$\lceil retracting \wedge bucket\_moving\_signal == finished \rceil \rightarrow \lceil retracted \rceil,$
Stab-12:	$\lceil bucket\_moving \wedge bucket\_moving\_signal == finished \rceil \rightarrow \lceil idle \rceil.$

Self-Adaptiveness of Robotic Systems. In robotic systems, self-adaptiv-eness plays a crucial role in enhancing responsiveness to dynamic environments. Self-adaptive systems can autonomously adapt and adjust their behavior and configuration in response to changing environments, uncertainties, or unseen conditions. To avoid damaging itself, we have ensured that our robotic arm system takes necessary measures if it goes out of safe range. The arm can damage itself by digging in a rocky area. To avoid this, we have used the current sensor and its temperature goes up when the arm keeps digging a same place. In this case, the robotic system either halts for a while or tries another place for digging.

6. Mapping duration calculus to PRISM

To formally verify the system in the PRISM model checker, we have mapped DC implementables to a PRISM model. The proposed mapping technique differs from the rule-based approach suggested by [29]. The DC implementables look like pseudo-code and can easily be mapped to the PRISM model by applying its syntax.

6.1 *Mapping states*

To map states, first, we collect all states specified in the DC implementables (e.g. Table 3) and define their corresponding initial states (labeled as Init). In Table 3, the bucket controller has three states: idle, bucket_moving, and current_protection.

To specify these three states in PRISM, we group them into one state variable as they represent the bucket controller's state. We name it controller_state, which contains all three states. The Controller's initial state is shown below:

Init-1: $\lceil\rceil \vee \lceil idle \rceil; true$

In PRISM, we declare the state variable and initialize it as follows:

```
//0: idle, 1: bucket_moving, 2: current_protection
controller_state: [0..2] init 0;
```

6.2 *Mapping channels*

Channels are used for communication between components. The bucket has finished, expand, stop, and retract channels. In PRISM, we group them as a set of controller_signals = {finished, expand, stop, retract}.

```
//0: finished, 1: expand, 2: retract, 3: stop
controller_signal: [0..3];
```

The channel facilitates state transactions. When the linear actuator expands and receives the signal of finished, it moves to the 'extended' state to indicate that the linear actuator has reached the 'extended' state. The DC implementable is as follows:

$$\text{Stab: } \lceil expanding \wedge bucket\_moving\_signal == finished \rceil \rightarrow \lceil extended \rceil$$

The following is the translated PRISM code from DC implementable:

```
[]linear_actuator_state = 1 & controller_signal = 0-> 1.0: (linear_actuator_state' = 3);
```

6.3 *Mapping sequences*

Sequence refers to state transactions without a guard. PRISM can assign a probability to a new state. If there is only one state transaction, it simply assigns 1.0 to that state. For example, once the Arm sub-system finishes the initial setup, it automatically transitions to an idle state.

"The states of the Arm's controller are set up as follows in PRISM:"

```
//0: setup, 1: idle, 2: operation,
//3: increment, 4: decrement,
//5: lower_limit, 6: upper_limit,
//7: halt
arm_controller_state: [0..7] init 0;
```

The controller transitions to idle once the setup is complete, as defined in DC implementables:

Seq-1: $\lceil setup \rceil \rightarrow \lceil idle \rceil$

The syntax in PRISM is as follows:

```
[]arm_controller_state = 0-> 1.0: (arm_controller_state' = 1);
```

Using DC specification, the non-deterministic sequences can also be specified. Following is an example of non-deterministic sequences where the controller can either move or go to the idle state. The model-checker non-deterministically picks the sequence.

Seq-1: $\lceil setup \rceil \rightarrow \lceil idle \rceil$,

Seq-2: $\lceil setup \rceil \rightarrow \lceil move \rceil$.

6.4 *Mapping transactions with guard*

PRISM uses commands to describe each module's behavior, comprising a guard and one or more updates. In the Bucket sub-system, when the Controller is idle, and its current is different than the Current Sensor's read current_t, the Controller moves to bucket_moving state. The DC implementable is described as follows:

Stab-1: $\lceil idle \wedge current \neq current\_t \rceil \rightarrow \lceil bucket\_moving \rceil$.

In PRISM, the transaction with the guard can be modeled as follows:

$$[]\texttt{controller\_state} = 0 \ \&\ (\texttt{current} != \texttt{current\_t}) -> 1.0 : (\texttt{controller\_state}' = 1);$$

In PRISM, $current \neq current\_t$ is a guard on transition. The controller_state's value is 0, which means the controller is in the setup state. The transition probability is 1.0, and then there will be a transition to controller_state = 1, which is the idle state.

7. Results and discussions

The findings of this paper are both accessible and valuable to readers outside the field of computer science, as the methodology simplifies complex concepts into practical steps. For example, formal specification and verification are clarified through the use of Duration Calculus and PRISM model checking, alongside intuitive examples like the mining robotic arm. By breaking down intricate concepts, such as probabilistic analysis and temporal logic, into real-world applications and clearly defined processes, the study bridges the gap between theory and practice. This approach ensures that professionals from diverse fields, such as robotics, automation, and systems engineering, can effectively apply these techniques to improve system reliability and safety, even if they lack deep expertise in formal methods. To explain the effectiveness of the methodology, we have presented and discussed the results in this section.

7.1 *Property verification*

PRISM [30] model checker is the tool for analyzing probabilistic systems through verifying specifications written in the PCTL and CSL. Non-probabilistic model checkers usually verify whether or not the design satisfies the requirements. A probabilistic model checker can answer the question further on how much of the probability of the specification meets the requirements. The PRISM runs the model and verifies the properties against it. Fixing the model and reiterating the verification process when the property is violated is simple. We performed the probabilistic analysis by defining the following properties.

The PRISM model checker uses the S operator to analyze steady-state behavior and the P operator to assess long-term event probabilities for the Arm system. Verifying the arm system design in PRISM evaluates state-based properties, including normal operation with over 99.9% probability of staying within the safety range and effective halting to prevent damage in abnormal motor conditions. Deadlock-freeness is verified with a 99.99% probability of the system being in a stable progressing state.

Table 4. Operators supported in PRISM

Operators	Description
<=>	if-and-only-if
=>	implication
?	condition evaluation: condition ? a : b means "if the condition is true then a else b"
P	probabilistic operator
S	steady-state operator
A	for-all operator
E	there-exists operator

Figures 5 and 6 show the execution time and memory consumption for each property verification. We have modeled the same robot using UPPAAL Stratego, and the results are compared with those of the PRISM model checker. The properties P1 and P2 take the highest execution time since they need to check all reachable states, which 99.9% of the time, the system operates in a normal range without reaching the halt state. P3 and P4 consume more memory due to their nature of traversing all possible cases to break the safety range.

Table 5. Temporal operators supported in PRISM

Operator	Description
X	Next
U	Until
F	Eventually (sometimes called "future")
G	Always (sometimes called "globally")
W	Weak until
R	Release

Table 6. Properties of the robotic arm system

Property No.	Properties	Properties in PCTL Syntax
P1	Arm doesn't halt.	$\mathscr{S} > 0.999\,[!\text{halt}]$
P2	Arm works in the safe range.	$\mathscr{S} > 0.999\,[\text{encoder\_recorded\_position} \leq \text{upper\_thresh} \wedge \text{encoder\_recorded\_position} \geq \text{lower\_thresh}]$
P3	Arm halts if it is out of safe range.	$\mathscr{S} > 0.999\,[(\text{encoder\_recorded\_position} > \text{upper\_thresh} \vee \text{encoder\_recorded\_position} < \text{lower\_thresh}) \leftrightarrow \text{halt}]$
P4	Bucket protects itself from damage when the current is too high.	$\mathscr{S} > 0.999\,[(\text{current} \geq \text{upper\_current}) \rightarrow \text{current\_protection}]$
P5	The system operates 99.9% normally without any deadlock.	$\text{filter}(\text{forall}, \text{"deadlockfree"} \Rightarrow \mathbb{P} \geq 0.999[\mathscr{F}\,!\text{halt}])$
P6	Probability of deadlock (halt meantime increment) is very low.	$\text{filter}(\max, \mathbb{P} =?[\mathscr{F}\,\text{halt} \wedge (\text{increment} \vee \text{decrement})]$

We verified the constraints listed in Table 6 in PRISM regarding the probability of a property and the maximum probability. We evaluate that the model defined from the formal specification can achieve 99.9% probabilities of

maintaining the system's liveness, consistent with the verification that 99.9% of the time, the Motor is spinning within the lower and upper thresholds.

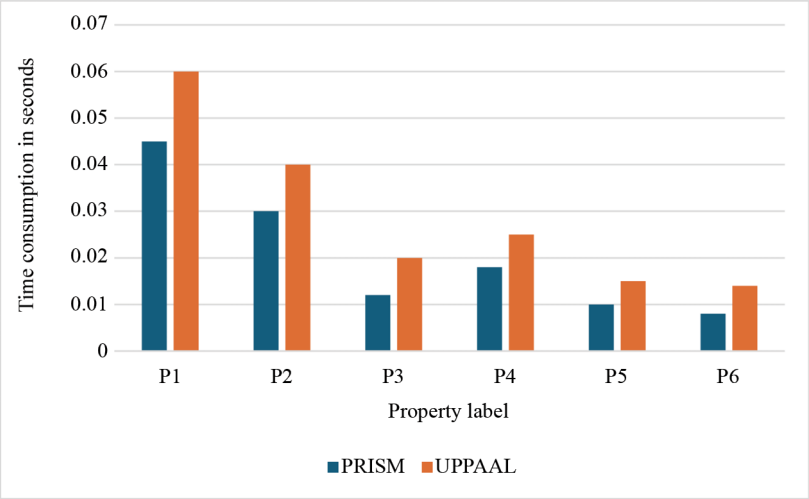

Figure 5. Time consumption

For safety, we check the probability of the system being halted when the Motor spins out of the safety range. The result returns a 99.9% of safety, fairness, and deadlock-freeness. We also examine the maximum probability of the system being deadlocked when the microcontroller halts the system, but the motor keeps spinning. The result shows that the maximum probability of system failure is near zero. The probability analysis for a more complex system can even predefine some runtime variables and constraints of related components' errors or faults, and then use PRISM to check the system's overall maximum system failure.

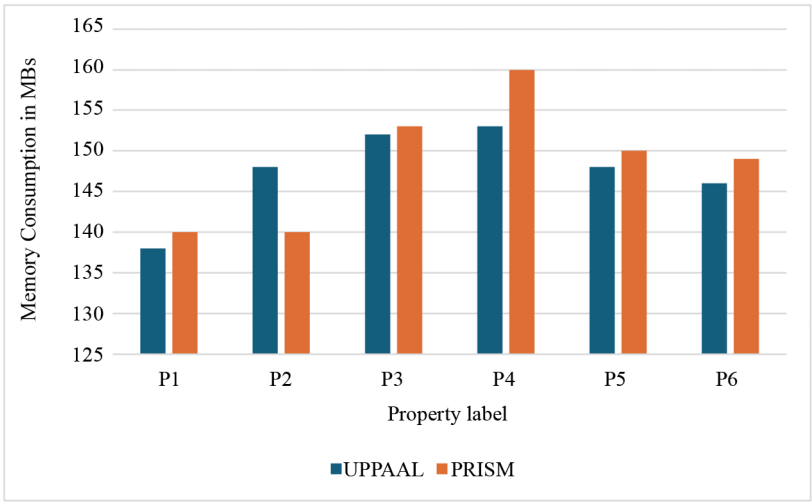

Figure 6. Memory consumption

Challenges Encountered in Modeling a Self-Adaptive System. Abstracting a self-adaptive system while maintaining essential behaviors for verification is challenging. Oversimplification may overlook critical adaptation behaviors,

whereas overly detailed models exacerbate the state-space problem. The model-checking tools also have some limitations. Existing model-checking tools are not always designed to handle the complexity of self-adaptive systems, especially when dealing with distributed architectures. Adaptation policies in self-adaptive systems often involve non-deterministic choices, making it difficult to verify all possible behaviors exhaustively. A significant issue with model checking is state space explosion, particularly for complex systems. Breaking the system into smaller, more manageable parts and independently verifying each is one efficient tactic. The system's computational overhead can be greatly decreased by modularizing it so that we can concentrate on confirming the accuracy of each component within clearly defined constraints. To maintain the entire system's integrity, these parts are gradually integrated after being validated, and their interactions are then rechecked. This divide-and-conquer strategy successfully reduces state space explosion while preserving the robustness of the verification process when paired with tools like PRISM for probabilistic model checking. Future work should focus on automated decomposition strategies and optimization techniques to improve scalability for large-scale, real-world systems with dynamic behaviors.

Implication of Research. The suggested methodology improves safety and dependability in self-adaptive systems using probabilistic verification and formal modeling to ensure correctness under uncertainty. Although a mining robotic arm was used for demonstration, this method can be widely applied to fields like autonomous cars, industrial automation, and healthcare robotics, where high-stakes operations and dynamic environments necessitate reliable and flexible systems. It provides a scalable framework for increasing operational efficiency and preventing failures across various applications by tackling issues like runtime uncertainties and design flaws.

8. Conclusions

The study focuses on the formal specification, modeling, and verification of a robotic system. It ensures coordinated and safe functionality for arm and bucket sub-systems, maintaining liveness, fairness, safety, and deadlock-free operation. Applied to an industrial robotic arm case study, the method ensures functional correctness and provides quantitative performance and reliability measures. The novel integrated methodology uses Duration Calculus (DC) for rigorous specification, a systematic mapping procedure to translate DC specifications into PRISM models. This approach minimizes gaps between specification and implementation, ensuring runtime correctness despite uncertainties. The robotic arm case study demonstrates its applicability to industrial systems with real-time and probabilistic aspects. Key aspects include expressive formal specification with DC, systematic mapping to PRISM, extensive verification of critical properties, and quantitative analysis of system robustness. The case study validates the methodology's effectiveness for real-world systems. Future research will focus on automated decomposition and optimization techniques to improve scalability for large-scale, real-world systems with dynamic behaviors. Moreover, a tool will be developed to automate the process of specification and analysis. The tool will help the practitioner quickly and efficiently analyze a self-adaptive system.

Conflict of interest

There are no conflicts of interest for this study.

References

[1] Kephart JO, Chess DM. The vision of autonomic computing. *Computer*. 2003; 36(1): 41-50. Available from: https://doi.org/10.1109/MC.2003.1160055.

[2] Oreizy P, Medvidovic N, Taylor RN. Architecture-based runtime software evolution. In: *Proceedings of the 20th International Conference on Software Engineering*. Kyoto, Japan: IEEE; 1998. p.177-186. Available from: https://doi.org/10.1109/ICSE.1998.671114.

[3] Cheng BHC, Eder KI, Gogolla M, Grunske L, Litoiu M, Müller HA, et al. Using models at runtime to address assurance for self-adaptive systems. In: Bencomo N, France R, Cheng BHC, Aßmann U. (eds.) *Models@run.time: Foundations, Applications, and Roadmaps*. Cham: Springer International Publishing; 2014. p.101-136. Available from: https://doi.org/10.1007/978-3-319-08915-7_4.

[4] Hezavehi SM, Weyns D, Avgeriou P, Calinescu R, Mirandola R, Perez-Palacin D. Uncertainty in self-adaptive systems: A research community perspective. *ACM Transactions on Autonomous and Adaptive Systems*. 2021; 15(4): 1-36. Available from: https://doi.org/10.1145/3487921.

[5] Hachicha M, Halima RB, Kacem AH. Formal verification approaches of self-adaptive Systems: A survey. *Procedia Computer Science*. 2019; 159: 1853-1862. Available from: https://doi.org/10.1016/j.procs.2019.09.357.

[6] Luckcuck M, Farrell M, Dennis LA, Dixon C, Fisher M. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys*. 2019; 52(5): 1-41. Available from: https://doi.org/10.1145/3342355.

[7] Cámara J, Troya J, Vallecillo A, Bencomo N, Calinescu R, Cheng BHC, et al. The uncertainty interaction problem in self-adaptive systems. *Software and Systems Modeling*. 2022; 21(4): 1277-1294. Available from: https://doi.org/10.1007/s10270-022-01037-6.

[8] Oreizy P, Medvidovic N, Taylor RN. Architecture-based runtime software evolution. In: *Proceedings of the 20th International Conference on Software Engineering*. Kyoto, Japan: IEEE; 1998. p.177-186. Available from: https://doi.org/10.1109/ICSE.1998.671114.

[9] Weyns D, Malek S, Andersson J. FORMS: A formal reference model for self-adaptation. In: *Proceedings of the 7th International Conference on Autonomic Computing (ICAC '10)*. Washington, DC, USA: Association for Computing Machinery; 2010. p.205-214.

[10] Liang H, Dong JS, Sun J, Wong WE. Software monitoring through formal specification animation. *Innovations in Systems and Software Engineering*. 2009; 5(4): 231-241. Available from: https://doi.org/10.1007/s11334-009-0096-1.

[11] Tarasyuk A, Pereverzeva I, Troubitsyna E, Latvala T, Nummila L. Formal development and assessment of a reconfigurable on-board satellite system. In: *Proceedings of the 31st International Conference on Computer Safety, Reliability, and Security (SAFECOMP'12)*. Magdeburg, Germany: Springer-Verlag; 2012. p.210-222. Available from: https://doi.org/10.1007/978-3-642-33678-2_18.

[12] Vardi MY. An automata-theoretic approach to linear temporal logic. In: Moller F, Birtwistle G. (eds.) *Logics for Concurrency: Structure versus Automata*. Berlin, Heidelberg: Springer; 1996. p.238-266. Available from: https://doi.org/10.1007/3-540-60915-6_6.

[13] Celaya JR, Desrochers AA, Graves RJ. Modeling and analysis of multi-agent systems using petri nets. In: *2007 IEEE International Conference on Systems, Man and Cybernetics*. Montreal, QC, Canada: IEEE; 2007. p.1439-1444. Available from: https://doi.org/10.1109/ICSMC.2007.4413960.

[14] Costelha H, Lima P. Modelling, analysis and execution of robotic tasks using petri nets. In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. San Diego, CA, USA: IEEE; 2007. p.1449-1454. Available from: https://doi.org/10.1109/IROS.2007.4399365.

[15] Halder R, Proença J, Macedo N, Santos A. Formal verification of ROS-Based robotic applications using timed-automata. In: *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. Buenos Aires, Argentina: IEEE; 2017. p.44-50. Available from: https://doi.org/10.1109/FormaliSE.2017.9.

[16] Iftikhar MU, Weyns D. A case study on formal verification of self-adaptive behaviors in a decentralized system. *Electronic Proceedings in Theoretical Computer Science*. 2012; 91: 45-62. Available from: https://doi.org/10.4204/EPTCS.91.4.

[17] Lyons DM, Arkin RC, Jiang S, Harrington D, Tang F, Tang P. Probabilistic verification of multi-robot missions in uncertain environments. In: *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*. Vietri sul Mare, Italy: IEEE; 2015. p.56-63. Available from: https://doi.org/10.1109/ICTAI.2015.22.

[18] Hoffmann R, Ireland M, Miller A, Gethin N, Veres S. Autonomous agent behaviour modelled in PRISM–A case study. *arXiv:1602.00646*. 2016. Available from: https://doi.org/10.48550/arXiv.1602.00646.

[19] Konur S, Dixon C, Fisher M. Formal verification of probabilistic swarm behaviours. In: *Swarm Intelligence*. Berlin, Heidelberg: Springer; 2010. p.440-447. Available from: https://doi.org/10.1007/978-3-642-15461-4_42.

[20] Celaya JR, Desrochers AA, Graves RJ. Modeling and analysis of multi-agent systems using petri nets. *Journal of Computers*. 2009; 4(10): 1439-1444. Available from: https://doi.org/10.1109/ICSMC.2007.4413960.

[21] Konur S, Dixon C, Fisher M. Formal verification of probabilistic swarm behaviours. In: *Swarm Intelligence*. Berlin, Heidelberg: Springer; 2010. p.440-447. Available from: https://doi.org/10.1007/978-3-642-15461-4_42.

[22] Konur S, Dixon C, Fisher M. Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems*. 2012; 60(2): 199-213. Available from: https://doi.org/10.1016/j.robot.2011.11.005.

[23] Massink M, Brambilla M, Latella D, Dorigo M, Birattari M. On the use of Bio-PEPA for modelling and analysing collective behaviours in swarm robotics. *Swarm Intelligence*. 2013; 7: 201-228. Available from: https://doi.org/10.1007/s11721-013-0083-x.

[24] Clarke EM. Model checking. In: Ramesh S, Sivakumar G. (eds.) *Foundations of Software Technology and Theoretical Computer Science*. Berlin, Heidelberg: Springer; 1997. p.54-56. Available from: https://doi.org/10.1007/BFb0058022.

[25] Olderog ER, Dierks H. *Real-Time Systems: Formal Specification and Automatic Verification*. 1st ed. USA: Cambridge University Press; 2008.

[26] Meyer R, Faber J, Rybalchenko A. Model checking duration calculus: A practical approach. In: Barkaoui K, Cavalcanti A, Cerone A. (eds.) *Theoretical Aspects of Computing - ICTAC 2006*. Berlin, Heidelberg: Springer; 2006. p.332-346. Available from: https://doi.org/10.1007/11921240_23.

[27] Mabu R, Al-Shaer E, Duan Q. A formal verification of configuration-based mutation techniques for moving target defense. In: *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Cham: Springer International Publishing; 2020. p.61-79. Available from: https://doi.org/10.1007/978-3-030-63086-7_5.

[28] Mabu R, Arif F. Translating activity diagram from duration calculus for modeling of real-time systems and its formal verification using UPPAAL and DiVinE. *Mehran University Research Journal of Engineering and Technology*. 2016; 35(1): 139-154. Available from: https://doi.org/10.22581/muet1982.1601.15.

[29] Charfi F, Ahmed S, Tahar S, Ayed LB. *An Approach Translating CoD Specification to be Checked by UPPAAL*. 2021. Available from: https://doi.org/10.13140/RG.2.2.18693.17128.

[30] Kwiatkowska M, Norman G, Parker D. PRISM: Probabilistic symbolic model checker. In: Field T, Harrison PG, Bradley J, Harder U. (eds.) *Computer Performance Evaluation: Modelling Techniques and Tools*. Berlin, Heidelberg: Springer; 2002. p.200-204. Available from: https://doi.org/10.1007/3-540-46029-2_13.