Proceedings of the 35th European Safety and Reliability & the 33rd Society for Risk Analysis Europe Conference
Edited by Eirik Bjorheim Abrahamsen, Terje Aven, Frederic Bouder, Roger Flage, Marja Ylönen
 Published by Research Publishing, Singapore.
doi: 10.3850/978-981-94-3281-3_ESREL-SRA-E2025-P6744-cd

Enhancing Resilience in Robotic Systems through Self-Awareness and Adaptive Recovery

Ruichao Wu

Robot and Assistive Systems Department, Fraunhofer Institute for Manufacturing Engineering and Automation (IPA) Germany.
E-mail: ruichao.wu@ipa.fraunhofer.de

Andrey Morozov

Institute of Industrial Automation and Software Engineering (IAS), University of Stuttgart, Germany.
E-mail: andrey.morozov@ias.uni-stuttgart.de

Björn Kahl

Robot and Assistive Systems Department, Fraunhofer Institute for Manufacturing Engineering and Automation (IPA) Germany.
E-mail: björn.kahl@ipa.fraunhofer.de

Robotic systems are becoming increasingly complex in both structure and behavior, integrating multiple components to provide advanced services. As this complexity grows, so does the likelihood of faults, making resilience essential for maintaining dependable functionality. This paper proposes a novel framework to enhance the resilience of robotic systems by enabling self-awareness and self-recovery in response to errors. To manage this complexity, we introduce the concept of a "Skill Chain, a set of components that collaboratively deliver specific services required for the system to transition through its states and achieve its mission goals. By continuously monitoring its internal state, the system detects errors before they propagate beyond the skill chain level. Upon detection, the system evaluates its current state and available resources. If spare components or other fault tolerance mechanisms exist, it reconfigures itself by forming a new skill chain capable of maintaining service delivery. Recovery strategies, such as backward recovery to return to a previously successful state or forward recovery to adapt service delivery, are dynamically applied to ensure minimal disruption.
The paper introduces the approach concept and demonstrates its applicability on exemplar robotic systems.

Keywords: Robotics, Reliability, Robustness, Resilience, Self-awareness, Recovery, Fault Containment.

1. Introduction

Robotic systems are advancing rapidly, requiring the integration of diverse hardware and software components to achieve complex tasks. However, as their structure and behavior grow in complexity, so does the risk of faults, which can disrupt functionality. Morozov and Janschek (2014). Therefore, ensuring fault tolerance, adaptability, and scalability are critical for maintaining dependable operations. Hancher and Hornby (2006); Jaekel et al. (2014); Zhang et al. (2017). Addressing these demands requires a structured approach to managing robotic systems, particularly in how tasks are executed and how faults are handled. Heuss et al. (2019); Murray (2013).

A key challenge in robotics lies in achieving modular, adaptable task execution while ensuring robust fault containment and enabling flexible error recovery. Existing solutions often address these issues independently, focusing either on task-specific solutions or static fault recovery mechanisms, which limits their applicability across different platforms and environments. Jacobsson et al. (2016); Vonásek et al. (2015); Ahmadzadeh et al. (2015). Few approaches provide a cohesive framework that integrates fault detection, system reconfiguration, and recovery strategies, and they fail to provide a unified framework that supports both modular task execution and dynamic fault handling across heterogeneous robotic platforms. Such limitations prompt the research question: *How can a framework support modular task execution, facilitate platform-independent operation, and ensure dynamic fault handling in a unified architecture?*

Contribution: This paper presents a novel framework designed to provide robotic systems with self-awareness and self-recovery capabilities, enhancing their resilience. The framework introduces the concept of *Skill Chains*, which are a set of software components that collaborate to deliver specific services. These services enable a robotic system to transition between states and accomplish mission objectives. By continuously monitoring its internal state, the system detects faults before they propagate, evaluates the status of skill chains, and apply appropriate recovery strategies. These include backward recovery (reverting to a prior successful state) and leveraging redundant components to

maintain service continuity. The proposed approach bridges the gap between modular task execution and system-wide fault tolerance, paving the way for more reliable and adaptable robotic systems in dynamic environments.

2. Related Work

Achieving resilience in robotic systems requires addressing fault tolerance, modularity, and adaptability. While existing approaches have made significant advancements in these areas, they often work in isolation, leading to limitations in system-wide resilience. There is still a need for an overarching framework that integrates these components and fosters robust, adaptable robotic systems.

2.1. *Modular Robotic Architectures*

Modern robotic architectures increasingly emphasize modularity. It enhances the manageability and scalability of robotic systems by dividing them into smaller, more manageable subsystems. Frameworks such as ROS (Robot Operating System) provide a foundation for developing modular robotic components by abstracting hardware interfaces and facilitating integration. Quigley et al. (2009). Based on ROS, researchers develop frameworks to address diverse operational requirements through reusable components. Ahmadzadeh et al. (2015) and scalable fulfillment systems. Wang et al. (2020). Domain-specific languages are proposed to enhance reliability and safety in robotics through modular programming approaches. Wigand et al. (2017); Beaulieu et al. (2018); Hammoudeh Garcia et al. (2019); Rizwan (2024). Additional work focuses on plug-and-play modularity to enhance dynamic adaptability in robotic platforms. Sell et al. (2019). These modular approaches reduce development time and improve overall system dependability. Our work advances these concepts by embedding fault detection and recovery solutions directly into the architecture.

2.2. *Fault-Tolerant Frameworks*

Robotic systems require effective fault tolerance to operate reliably in dynamic and unpredictable settings. Recent work addresses this need through fault diagnosis, dynamic recovery, and adaptive control strategies. Prorok et al. (2021). Dynamic reconfiguration methods, such as using declarative models, allow systems to adapt to faults in real time, improving fault tolerance and operational continuity. Nordmann et al. (2021). Despite these approaches, neural network-based fault diagnosis combined with reinforcement learning for dynamic fault compensation enables more precise and context-aware fault handling. Yan et al. (2022). Deep learning-based approaches allow robots to response adaptively to diverse failures, such as in navigation systems where they maintain operations despite hardware or software disruptions. Gültekin et al. (2022) Meanwhile, unbiased active inference controllers leverage sensory prediction errors and probabilistically robust thresholds to improve state estimation accuracy and reduce false positives in fault detection. Baioumy et al. (2021). Additional, robust control techniques, such as fixed-time sliding mode control, have demonstrated improved fault recovery for robotic manipulators. Van and Ceglarek (2021); Milecki and Nowak (2023). Architectural advancements also play a key role in fault tolerance. Self-maintenance architectures, such as ReFrESH, enable robots to autonomously mitigate faults in multi-robot systems. Cui et al. (2014). Hierarchical architectures integrate fault detection and recovery across multiple system levels, providing a more structured approach to managing errors. Ahn et al. (2010). While these methods improve fault handling, they often target specific subsystems rather than addressing system-wide fault resilience.

2.3. *Behavior-Tree-Based Control for Resilience*

Behavior trees (BTs) are a modular and hierarchical framework, originating from the gaming industry. BTs provide a structured representation of robot behaviors using nodes to represent actions, conditions, and sequences. They improve scalability and reusability compared to finite state machines (FSMs). Colledanchise and Ögren (2018). Current studies showcase the versatility of BTs in robotic control. Iovino et al. (2022). Dynamic adaptation of BTs enables robots to modify their behavior at runtime, responding to environmental changes and maintaining operational continuity. BTs have been effectively used in safety supervision systems to ensure functional safety in autonomous robots. Conejo et al. (2024). In addition, combining BTs with reinforcement learning enhances decision-making and execution. Pezzato et al. (2023). Collaborative robotic tasks benefit from BTs, where fault-tolerant behaviors ensure task completion despite disruptions. Akkaladevi et al. (2024). Despite these advances, current implementations of BTs often focus on specific domains or tasks, lacking a generalized framework for system-wide behavior control and error recovery. BT integration with advanced diagnostic systems and modular architectures is still underexplored. Our work aims to address these gaps by developing a comprehensive behavior-tree-based framework that integrates fault diagnosis, adaptive recovery, and modularity to improve resilience across robotic systems.

3. Framework Architecture for Modular and Fault-Tolerant Robotic Systems

Robotic systems require the integration of diverse hardware and software components to deliver advanced services. For instance, a robotic system designed for simple assembly tasks involves hardware components like a camera, a manipulator, and a gripper, alongside software components such as a camera driver, a manipulator driver, a gripper driver, and algorithms for object detection, collision-free motion planning, inverse kinematics (IK), and trajectory execution. The interaction between these components leads to a complex system that demands a modular and structured approach.

To address these challenges, we propose a framework that decomposes robotic systems into manageable subsystems using *Skill Chains*, *Skill Clients*, a *Control Flow Operator* and a *Behavior Tree-based Control Flow* as shown in Fig. 1. This layered structure simplifies the integration of new components. It also enables real-time error detection and localized recovery strategies. As a result, the robotic system maintains its overall functionality, even when certain parts fail or require updates.

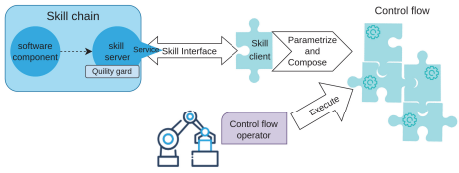

Fig. 1. Framework architecture for modular and fault-tolerant robotic systems, including Skill Chains, Skill Clients, Control Flow Operator, and Behavior Tree-based Control Flows.

3.1. *Skill Chains and Skill Clients*

A **Skill Chain** $\mathcal{S}_j$ is a modular group of software components $\{C_{j1}, C_{j2}, \ldots, C_{jm}\}$ that collaboratively deliver a specific service $\mathcal{Q}_j$. Formally, it is defined as:

$$\mathcal{S}_j = \{C_{j1}, C_{j2}, \ldots, C_{jm}\}, \text{where } \mathcal{Q}_j = F(\mathbf{C}_j, Q_{\text{input}}).$$

Here:

- $\mathbf{C}_j = \{C_{j1}, C_{j2}, \ldots, C_{jm}\}$ is the set of components within the skill chain $\mathcal{S}_j$.
- Q_{input} and $Q_{\text{output}} = \mathcal{Q}_j$ represent the quality of input and output data, respectively.
- F is a function that represents the collaborative processing performed by the components to transform Q_{input} into the desired service $\mathcal{Q}_j$.

Each component C_{ji} performs a specific function and provides outputs required as inputs for the next. This concept keeps the system modular and allows developers to adapt or replace individual parts without disrupting the entire architecture.

At the end of each skill chain is a **Skill Server**, which acts as the final processing entity. This server provides the requested service $\mathcal{Q}_j$ through a **Skill Interface** and confirms that performance metrics meet a preset quality threshold. These metrics might involve execution time or accuracy, and each metric carries a weight. The Skill Server aggregates these metrics into a single quality measure:

$$Q_{\text{service}} = \sum_{i=1}^{n} w_i \cdot f(M_i),$$

where M_i represents the i-th quality metric being monitored, w_i is the weight assigned to the i-th metric, reflecting its importance, and $f(M_i)$ is a function that evaluates whether the i-th metric is satisfied. The function $f(M_i)$ is defined as:

$$f(M_i) = \begin{cases} 1, & \text{if the metric is satisfied,} \\ 0, & \text{otherwise.} \end{cases}$$

The Skill Server ensures that the quality of service (QoS) satisfies a predefined threshold:

$$Q_{\text{service}} \geq Q_{\text{threshold}}.$$

If the QoS falls below this threshold, the Skill Server notifies the Skill Client of the failure and the control flow will be halted.

The **Skill Client** sends service requests to the Skill Server with parameters specifying the desired service and any task-specific configurations. It acts as the link between the high-level application flow (the control flow) and the Skill Chain. This design makes each request more dynamic because the system can adjust parameters or switch to alternative Skill Chains when tasks or conditions change.

3.2. *Behavior Tree-based Control Flow*

The **Control Flow** represents the application logic of the robotic system and is implemented using a *Behavior Tree*. BTs are a hierarchical model that can organize tasks into a tree-like structure, where each node represents an action, condition, or controller (e.g., sequence, fallback). This approach offers several advantages: it encapsulates each task or behavior as a modular node, enabling reusability and

easier reorganization; it allows complex tasks to be decomposed into subtasks, improving scalability and manageability; moreover, it evaluates conditions at runtime, allowing the robot to react or adapt if an error arises. The control flow operator parses the control flow and orchestrates the execution of skill chains by determining the sequence of service requests based on the application logic. When an application is running, the framework operates as follows:

(1) The control flow determines the sequence of tasks to execute based on the application's requirements.
(2) Parameterized skill client nodes in the control flow send service requests to their respective skill servers, triggering the execution of associated skill chains.
(3) Each skill chain processes the request, orchestrating its components to deliver the required service via the Skill Server. During execution, the server checks that the results meet the required standards. If a service fails, the Skill Server notifies the Skill Client, which updates its state and informs the control flow operator; the control flow operator halts the current control flow.
(4) The control flow resumes once the issue is resolved or an appropriate recovery mechanism is executed.

4. Fault Containment and Recovery Mechanisms

The proposed framework handles these faults by limiting their propagation and offering multiple recovery methods. In doing so, it preserves overall system reliability and supports flexible adaptation.

4.1. *Error Propagation and Containment*

Error propagation is an inherent phenomenon in software systems, where faults within a component can lead to errors that propagate to other interconnected components. Avizienis et al. (2004). If errors are not contained, this can result in cascading failures that affect the overall functionality of the system. The proposed framework is designed to contain error propagation strictly within the Skill Chain, ensuring that errors do not escalate into the higher-level control flow.

As shown in Fig. 2, error propagation in the framework occurs in two stages. Internal propagation takes place when a fault within a component, such as a software bug or unexpected input, leads to an incorrect internal state. This incorrect state then manifests as an error in the component's output. External propagation happens when the erroneous output from one component is received as an external fault by the next component, potentially causing further errors if the fault is not detected or addressed.

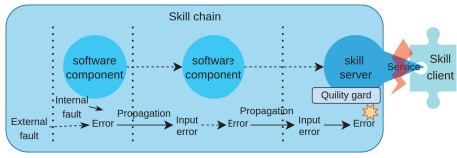

Fig. 2. Stages of error propagation and containment within the proposed framework. Errors originating within a component are either contained at the skill chain level or detected by the Skill Server before they escalate into the control flow. Avizienis et al. (2004).

Error propagation continues until the error reaches the Skill Server, which monitors QoS. If the QoS falls short of set standards, the Skill Server halts the skill chain and notifies the Skill Client. This localized response stops the error before it can disrupt the entire system.

4.2. *Recovery Mechanisms in the Framework*

The modular structure of this framework supports multiple recovery strategies. By breaking the system into skill chains, each with its own service interface, the framework allows both localized fixes and broader system-level adaptations. The conceptual recovery mechanisms include the following:

Reforming Skill Chains The modularity of skill chains enables the framework to reform a skill chain when a component fails. If spare components capable of producing the same type of output are available, the skill chain can dynamically replace the faulty component with an alternative. This capability ensures that the service provided by the skill chain continues without requiring a complete system halt. Reforming skill chains focuses on leveraging redundancy within the system to recover locally from faults.

Switching to Alternate Skill Chains The framework allows a skill client to switch to a different skill chain if the current skill chain fails to deliver the required service. Alternate skill chains providing the same service through the same skill interface can be utilized seamlessly. This mechanism leverages pre-existing redundancy and avoids halting the control flow. It also safeguards the system from single points of failure.

Backward Recovery In this approach, the framework rolls back to a previous checkpoint. These checkpoints act as intermediate recovery points that the system can roll back to

in the event of a failure. When a fault surfaces, the system reverts to the most recent stable state and resumes execution from there. This method minimizes the impact of faults and prevents errors from propagating beyond the checkpoint.

Forward Recovery Forward recovery focuses on adapting the system to bypass faults and continue operation. Unlike backward recovery, which reverts to a previous state, forward recovery dynamically reconfigures the control flow to ensure task completion. The framework supports this strategy through **Recovery Control Flow Generation**. When the existing control flow cannot accommodate a fault, the framework generates a recovery control flow composed of skill chains or recovery-specific actions. This new control flow addresses the fault by replacing faulty components with alternatives, adjusting task parameters to work around the fault, or executing alternative strategies to achieve the desired outcome. Recovery Control Flow Generation leverages existing skill chains and resources to execute forward recovery strategies dynamically and flexibly.

By offering these complementary techniques, the framework ensures that robotic systems can withstand a range of failures. Each mechanism targets faults at different levels of severity or scope. Consequently, the overall system remains operational and retains the flexibility to evolve with changing requirements.

5. Experimental Setup and Results

The experiments are conducted to validate the framework's capability to handle modular and reusable skill chains, demonstrate fault containment and recovery, and accommodates different robotic platforms under shared control flows. For these tests, we selected two industrial robotic arms: the PILZ PRBT 6 (PRBT) and the Universal Robots UR5e (UR5e). Each robot is configured to perform a *Pick and Place* task using the proposed framework. The two robotic setups used in the experiments are depicted in Fig. 3 and Fig. 4.

For the *Pick and Place* task, the system receives the object position and the place position as inputs. With this information, the system plans and executes a collision-free trajectory to pick the object from its provided position. Once the object is grasped, the system replans and executes another trajectory to place the object at the target location. Although the robots differ in hardware, they both rely on the same shared control flow and parameterized skill chains which adjust to each robots kinematic properties.

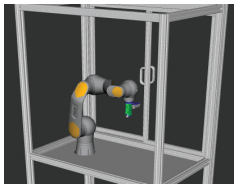

Fig. 3. PRBT Robot Setup: PRBT robot with Schunk EGP 50 gripper

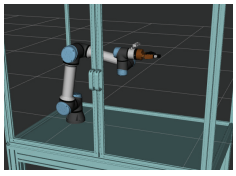

Fig. 4. UR5e Robot Setup: UR5e robot with Robotiq gripper.

5.1. *Modular Design of Skill Chains*

The concept of skill chains is exemplified in the Trajectory Execution Skill Chain for the UR5e robot, shown in Fig. 6. This skill chain consists of modular components that collaboratively deliver the trajectory execution service. This chain includes components from ROS [a] open-source community such as the Planning Scene Monitor and Robot State Publisher (light blue circles in Fig. 6) and a custom Skill Server (dark blue circle in Fig. 6). The Trajectory Execution Skill Server is a custom component which not only delivers the trajectory execution service but also monitors QoS, and handles specific tasks like monitoring execution time, collision detection, trajectory validation, and stop signal handling.

5.2. *Reusable Skill Chains Across Robots*

The same skill chain structure was adapted for the PRBT arm (Fig. 7). While the overall structure of the skill chain is the same, differences arise in the software components due to hardware-specific requirements. For instance, PRBT uses a unique driver and robot description to match its kinematics and dynamics, shown as orange circles in Fig. 7. Despite these variations, the ROS components (light blue circles) remain consistent across setups, demonstrating that the framework can easily switch between different robot models without redesigning the entire skill chain.

5.3. *Redundant Skill Chains*

The framework supports redundant skill chains to ensure flexibility and fault tolerance during task execution. For instance, the Motion Planning Skill Chain includes multiple planners, such as the Rapidly-exploring Random Tree* (RRT*) planner from the Open Motion Planning Library (OMPL)[b] and the Point-to-Point Planner (PTP) from Pilz

[a] https://docs.ros.org/en/jazzy/index.html
[b] `https://ompl.kavrakilab.org/`

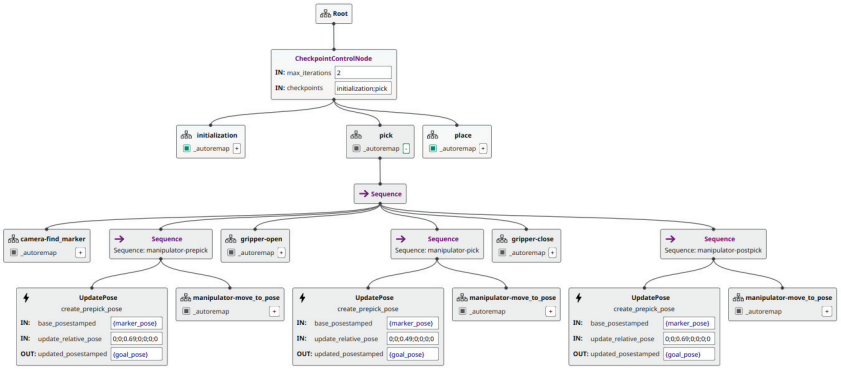

Fig. 5. Control flow with checkpoint-based backward recovery for the pick and place task.

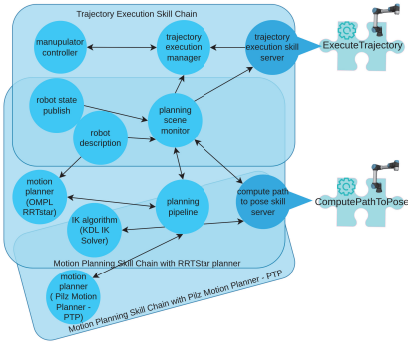

Fig. 6. Trajectory Execution Skill Chain for the UR5e robot. Light blue circles are ROS components; the dark blue circle is the custom skill server.

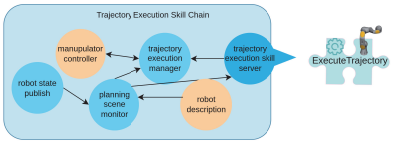

Fig. 7. Trajectory Execution Skill Chain for the PRBT robot. Light blue circles are ROS components; the dark blue circle is the custom skill server.

Industrial Motion Planner [c], as shown in Fig. 6.

In the UR5e's control flow (Fig. 8), the skill client "ComputePathtoPose" of the Motion Planning Skill Chain sends a request to the Skill Server with target pose and execution parameters. The control flow operator monitors the state of this skill client. If the primary planner fails, the fallback mechanism ensures that an alternate planner is invoked, maintaining task execution without manual intervention.

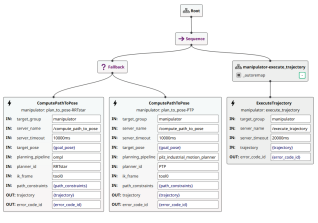

Fig. 8. Behavior tree control flow using redundant Motion Planning Skill Chain for the UR5e robot.

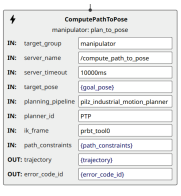

Fig. 9. Parameterized "ComputePath-toPose" skill client for PRBT robot.

5.4. *Shared Control Flows with Parameterized Skill Clients*

Both the PRBT and UR5e executes the *Pick and Place* task using the same control flow (Fig. 5), which was parameterized to accommodate hardware-specific requirements. For example, the PRBT robot uses a different IK frame name due to its distinct kinematic model compared to the UR5e robot(Fig. 9). This flexible parameterization confirms that the system can reuse control flows across multiple robots, streamlining deployment and maintenance.

5.5. *Fault Containment Capability*

The framework demonstrate fault containment by isolating errors within skill chains. For example, if the motion

[c] https://github.com/PilzDE/pilz_industrial_motion

planner failed to compute a trajectory, the Skill Server catches the error and prevents the error from affecting the control flow. Then the system can trigger recovery mechanismpotentially switching to another planner, as outlined in Sec. 5.3. This localized fault handling ensured that only the Skill Chain in question required correction, preserving the rest of the control flow.

5.6. *Backward Recovery Capability*

A checkpoint-based approach for rollback recovery is implemented using a *CheckPointControlNode*. It is a customized control node that reverts the control flow to the last successful checkpoint and retry the task when failures occur. The logic is detailed in algorithm 1. This mechanism ensured robust recovery while adhering to predefined retry limits. Fig. 5 illustrates the use of the *CheckPointControlNode* in the control flow for the pick-and-place application.

6. Conclusion

This work presents a modular architecture that introduces the *Skill Chain*, the *Skill Client*, the *Control Flow Operator* and the *Behavior Tree-based control flow* to manage the complexity of modern robotic applications, contain faults, and enable robust recovery. By decomposing tasks into smaller, reusable units, the framework helps to structure robotic task, easy adapt to diverse hardware platforms. Experiments on PRBT and UR5e robots demonstrate that the proposed solution supports platform-independent deployment, recovers dynamically from faults, and maintains operational efficiency. The system achieves fault containment by isolating errors within each skill chain, preventing them from affecting that high-level control flow. The integration of backward recovery mechanisms, such as reverting to previous checkpoints to restore functionality, ensures continuity of service even in the presence of failures. Future work will focus on exploring forward recovery techniques that dynamically adjust task execution in response to real-time faults. This expansion aims to reinforce adaptability of the robotic system and uphold consistent performance in varied operational settings.

Acknowledgement

This research is sponsored by the Baden-Württemberg Ministry of Economic Affairs, Labour and Tourism Baden-Württemberg at the AI Innovation Center "Learning Systems and Cognitive Robotics".

Input: List of checkpoints:
`checkpoints` $= \{A, B, C, \dots\}$
Input: Maximum allowed retries:
`max_iterations`
Output: `SUCCESS`, `FAILURE`
Initialize all nodes to `IDLE`;
Initialize `current_checkpoint_index` $\leftarrow 0$;
Set `last_checkpoint` $\leftarrow$ None;
Set `iteration_count` $\leftarrow 0$;
while *`iteration_count` $<$ `max_iterations`*
do
  Tick `current_node`;
  if *`current_node` returns `RUNNING`* **then**
    Return `RUNNING`;
  else if *`current_node` returns `SUCCESS` and `current_node` is a checkpoint* **then**
    Update `last_checkpoint` $\leftarrow$ `current_node`;
    Increment `current_checkpoint_index`;
    Proceed to the next node;
  else if *`current_node` returns `FAILURE`* **then**
    Revert to the last checkpoint: `current_checkpoint_index` $\leftarrow$ `index(last_checkpoint)`;
    Increment `iteration_count`;
    if *No valid checkpoint remains* **then**
      Revert to the checkpoint before `last_checkpoint`;
      if *All checkpoints exhausted* **then**
        Return `FAILURE`;
      end
    end
end
if *`iteration_count` $\geq$ `max_iterations`*
then
  Return `FAILURE`;
else
  Return `SUCCESS`;
end

Algorithm 1: Logic of CheckPointControlNode

References

Ahmadzadeh, H., E. Masehian, and M. Asadpour (2015). Modular robotic systems: Characteristics and applications. *Journal of Intelligent & Robotic Systems 81*, 317 – 357.

Ahn, H., D.-S. Lee, and A. Sang Chul (2010). A hierarchical fault tolerant architecture for component-based service robots. In *2010 8th IEEE International Conference on Industrial Informatics*, pp. 487–492.

Akkaladevi, S. C., M. Propst, K. Deshpande, M. Hofmann, A. Pichler, P. Sapoutzoglou, A. Zacharia, D. Kalogeras, and M. Pateraki (2024). Dynamic adaptability in human-robot col-

laboration for industrial assembly: A behaviour tree based task execution. In *International Conference on Flexible Automation and Intelligent Manufacturing*, pp. 305–312. Springer.

Avizienis, A., J.-C. Laprie, B. Randell, and C. Landwehr (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*(1), 11–33.

Baioumy, M., C. Pezzato, R. Ferrari, C. H. Corbato, and N. Hawes (2021). Fault-tolerant control of robot manipulators with sensory faults using unbiased active inference. In *2021 European Control Conference (ECC)*, pp. 1119–1125.

Beaulieu, A., S. N. Givigi, D. Ouellet, and J. T. Turner (2018). Model-driven development architectures to solve complex autonomous robotics problems. *IEEE Systems Journal 12*(2), 1404–1413.

Colledanchise, M. and P. Ögren (2018). *Behavior trees in robotics and AI: An introduction*. CRC Press.

Conejo, C., V. Puig, B. Morcego, F. Navas, and V. Milan'es (2024). Behavior trees in functional safety supervisors for autonomous vehicles.

Cui, Y., J. Lane, R. Voyles, and A. Krishnamoorthy (2014). A new fault tolerance method for field robotics through a self-adaptation architecture. In *2014 IEEE International Symposium on Safety, Security, and Rescue Robotics (2014)*, pp. 1–6.

Gültekin, Ö., E. Cinar, K. Özkan, and A. Yazc (2022). Multisensory data fusion-based deep learning approach for fault diagnosis of an industrial autonomous transfer vehicle. *Expert Systems with Applications 200*, 117055.

Hammoudeh Garcia, N., M. Lüdtke, S. Kortik, B. Kahl, and M. Bordignon (2019). Bootstrapping mde development from ros manual code - part 1: Metamodeling. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pp. 329–336.

Hancher, M. and G. Hornby (2006). A modular robotic system with applications to space exploration. In *2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)*, pp. 8 pp.–132.

Heuss, L., A. Blank, S. Dengler, G. Zikeli, G. Reinhart, and J. Franke (2019). Modular robot software framework for the intelligent and flexible composition of its skills. In *Advances in Production Management Systems. Production Management for the Factory of the Future*, IFIP Advances in Information and Communication Technology, pp. 248–256. Springer New York LLC.

Iovino, M., E. Scukins, J. Styrud, P. Ögren, and C. Smith (2022). A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems 154*, 104096.

Jacobsson, L., J. Malec, and K. Nilsson (2016). Modularization of skill ontologies for industrial robots. In *Proceedings of ISR 2016: 47st International Symposium on Robotics*, pp. 1–6.

Jaekel, S., M. Stelzer, and H.-J. Herpel (2014). Robust and modular on-board architecture for future robotic spacecraft. In *2014 IEEE Aerospace Conference*, pp. 1–11.

Milecki, A. and P. Nowak (2023). Review of fault-tolerant control systems used in robotic manipulators. *Applied Sciences 13*(4).

Morozov, A. and K. Janschek (2014). Probabilistic error propagation model for mechatronic systems. *Mechatronics 24*(8), 1189–1202.

Murray, L. (2013). *Fault Tolerant Morphogenesis in Self-reconfigurable Modular Robotic Systems*. Ph. D. thesis, University of York.

Nordmann, A., R. Lange, and F. M. Rico (2021). System modes - digestible system (re-)configuration for robotics. In *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, pp. 19–24.

Pezzato, C., C. H. Corbato, S. Bonhof, and M. Wisse (2023). Active inference and behavior trees for reactive action planning and execution in robotics. *IEEE Transactions on Robotics 39*(2), 1050–1069.

Prorok, A., M. Malencia, L. Carlone, G. S. Sukhatme, B. M. Sadler, and V. R. Kumar (2021). Beyond robustness: A taxonomy of approaches towards resilient multi-robot systems. *ArXiv abs/2109.12343*.

Quigley, M., K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Berger, R. Wheeler, and A. Y. Ng (2009). Ros: an open-source robot operating system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.

Rizwan, M. (2024). *Programming for Reliability and Safety in Robotics: The Role of Domain-Specific Languages: Domain Specific Programming for Safe and Reliable Robots*. Licentiate thesis, Lund University.

Sell, R., E. Väljaots, T. Pataraia, and E. Malayjerdi (2019). Modular smart control system architecture for the mobile robot platform. *Proceedings of the Estonian Academy of Sciences*.

Van, M. and D. Ceglarek (2021). Robust fault tolerant control of robot manipulators with global fixed-time convergence. *Journal of the Franklin Institute 358*(1), 699–722.

Vonásek, V., S. Neumann, D. Oertel, and H. Wörn (2015). Online motion planning for failure recovery of modular robotic systems. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1905–1910.

Wang, W., Y. Wu, J. Zheng, and C. Chi (2020). A comprehensive framework for the design of modular robotic mobile fulfillment systems. *IEEE Access 8*, 13259–13269.

Wigand, D., A. Nordmann, N. Dehio, M. Mistry, and S. Wrede (2017, December). Domain-specific language modularization scheme applied to a multi-arm robotics use-case. *Journal of Software Engineering for Robotics 8*(1), 45–64.

Yan, Z., J. Tan, B. Liang, H. Liu, and J. Yang (2022). Active fault-tolerant control integrated with reinforcement learning application to robotic manipulator. In *2022 American Control Conference (ACC)*, pp. 2656–2662.

Zhang, T., W. Zhang, and M. Gupta (2017). Resilient robots: Concept, review, and future directions. *Robotics 6*(4).