Comprehensive Adaptive Middleware for Mixed-Critical Cyber-Physical Networks

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Informatik und Mathematik
der Johann Wolfgang Goethe-Universität
in Frankfurt am Main

von

Melanie Feist

aus Friedrichshafen

Frankfurt 2023
(D30)

vom Fachbereich Informatik und Mathematik der
Johann Wolfgang Goethe-Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Martin Möller
Gutachter:	Apl. Prof. Dr. Mathias Pacher
	Prof. Dr. Lars Hedrich
	Prof. Dr. Roman Obermaisser
Datum der Disputation:	17.10.2023

Rückseite Titel

Abstract

Cyber Physical Systems (CPS) are growing more and more complex due to the availability of cheap hardware, sensors, actuators and communication links. A network of cooperating CPSs (CPN) additionally increases the complexity. This poses challenges as well as it offers chances: the increasing complexity makes it harder to design, operate, optimize and maintain such CPNs. However, on the other side an appropriate use of the increasing resources in computational nodes, sensors, actuators can significantly improve the system performance, reliability and flexibility. Therefore, self-X features like self-organization, self-adaptation and self-healing are key principles for such systems. Additionally, CPNs are often deployed in dynamic, unpredictable environments and safety-critical domains, such as transportation, energy, and healthcare. In such domains, usually applications of different criticality level exist. In an automotive environment for example, the brake has a higher criticality level regarding safety as the infotainment. As a result of mixed-criticality, applications requiring hard real-time guarantees compete with those requiring soft real-time guarantees and best-effort application for the given resources within the overall system. This leads to the need to accommodate multiple levels of criticality while ensuring safety and reliability, which increases the already high complexity even more.
This thesis deals with the question on how to conveniently, effectively and efficiently handle the management and complexity of mixed-critical CPNs (MC-CPNs). Since this cannot be done by the system developer without the assistance of the system itself any longer, it is essential to develop new approaches and techniques to ensure that such systems can operate under a range of conditions while meeting stringent requirements. Based on five research hypothesis, this thesis introduces a *comprehensive adaptive mixed-criticality supporting middleware for Cyber-Physical Networks* (*Chameleon*), which efficiently and autonomously takes care of the management and complexity of CPNs with regard to the mixed-criticality aspect. *Chameleon* contributes to the state-of-art by introducing and combining the following concepts:

- A comprehensive self-adaption mechanism on all levels of the system model is provided.
- This mechanism allows a flexible combination of parametric and structural adaptation actions (relocation, scheduling, tuning, ...) to modify the behavior of the system.
- Real-time constraints of mixed-critical applications (hard real-time, soft real-time, best-effort) are considered in all possible adaptation conditions and actions by the use of the *importance* parameter.

- CPNs are supported by the introduction of different scopes (local, system, global) for the adaptation conditions and actions. This also enables the combination of different scopes for conditions and actions.
- The realization of the adaptation with a MAPE-K loop instantiated by a distributed LCS allows for real-time capable reasoning of adaptation actions which also works on resource-spare systems.
- The developed rule language *Rango* offers an intuitive way to specify an initial rule set for LCS in the context of CPS/CPNs and supports the system administrators in the process of rule set generation.

To evaluate *Chameleon*, an extensive evaluation has been conducted which demonstrates the validity of the posed five research hypotheses for the design and development of *Chameleon*.

Keywords— Middleware, Cyber-Physical Networking, Cyber-Physical Systems, Mixed-Criticality, Self-X Properties, (autonomous) Adaptation

Zusammenfassung

Motivation Das rasante Wachstum von Cyber-Physical Systems (CPS) aufgrund der Verfügbarkeit von preiswerter Rechenhardware, Sensoren, Aktoren und Kommunikationsverbindungen hat zu einer starken Erhöhung der Systemkomplexität geführt. Im Automobilbereich z.B. ist die Anzahl und Leistungsfähigkeit dieser Komponenten in einem Fahrzeug in den letzten Jahren stetig gestiegen. Die Komplexität entsteht durch die Interdependenz von physikalischen Prozessen, Sensorik und Aktorik, Kommunikationsnetzen und Rechenressourcen. Die Kombination dieser Faktoren führt zu einer Vielzahl von Problemen, die gelöst werden müssen, z. B. Aufgabenzuweisung, Zeitplanung, Fehlertoleranz und Kommunikation. Mit der zunehmenden Anzahl heterogener Komponenten in einem solchen System steigt auch die Komplexität der Wechselwirkungen zwischen ihnen. Ein Netzwerk von kooperierenden CPS - ein sogenanntes Cyber-Physical Network (CPN) - macht die Situation noch komplexer, da Informationen zwischen den CPS im Netzwerk ausgetauscht und Aktionen koordiniert und eingesetzt werden müssen. Ein Beispiel für ein solches CPN sind kooperierende oder platoonierende Fahrzeuge, bei denen mehrere autonome Fahrzeuge in einem Konvoi mit geringen Abständen fahren und diese die Kommunikation zwischen den Fahrzeugen über IEEE 802.11p koordinieren.

CPNs werden häufig in dynamischen und unvorhersehbaren Umgebungen eingesetzt, insbesondere in sicherheitskritischen Bereichen wie Verkehr und Gesundheitswesen. In solchen Bereichen gibt es oft Anwendungen mit unterschiedlichem Kritikalitätsgrad. Beispielsweise ist die Bremse in einem Fahrzeug sicherheitskritischer als das Infotainment-System. Aufgrund dieser gemischten Kritikalität konkurrieren harte Echtzeit-Anwendungen mit solchen, die weiche Echtzeitgarantien oder Best-Effort-Behandlung erfordern. Dies führt dazu, dass mehrere Kritikalitätsebenen berücksichtigt werden müssen um Sicherheit und Zuverlässigkeit zu gewährleisten, was die ohnehin schon hohe Komplexität noch weiter steigert. Dieses hohe Maß an Komplexität erschwert die Verwaltung und Optimierung der Leistung, Zuverlässigkeit und Sicherheit solcher gemischt kritischen (mixed-critical) CPN (MC-CPN). Daher ist es unerlässlich, neue Ansätze und Techniken zu entwickeln, um den Betrieb von MC-CPN unter variierenden Bedingungen gemäß den gegebenen Anforderungen effizient, skalierbar und zuverlässig zu gewährleisten.

Die Autonomie, also die Fähigkeit eines Systems, sich selbstorganisierend zu betreiben und zu kontrollieren, spielt hierbei eine wichtige Rolle. Autonome Mechanismen und Techniken, die als Selbst-X-Eigenschaften (Selbstanpassung, Selbstorganisation, Selbstheilung, ...) bezeichnet werden, sind eine wichtige und wesentliche Methode,

um den Betrieb komplexer Systeme effektiv und effizient zu beherrschen. Daher ist die Entwicklung neuer autonomer Mechanismen und Techniken zur Verwaltung von MC-CPNs ein vielversprechender Ansatz, um deren Leistung, Effizienz, Zuverlässigkeit und Flexibilität zu verbessern.

Ansatz Als Grundlage der Arbeit werden zunächst eine Reihe von Forschungshypothesen aufgestellt:

Hypothese 1:
Die Integration von autonomem Verhalten durch Selbst-X Eigenschaften (z.B. Selbstanpassung, Selbstorganisation, ...) stellt einen geeigneten Ansatz dar, um das Management und die Komplexität von MC-CPNs zu bewältigen.

Hypothese 2:
Die Implementierung autonomer Mechanismen kann in einem Middleware-Framework gekapselt werden.

Hypothese 3:
Eine regelbasierte MAPE-K-Rückkopplungsschleife ist ein praktikabler Adaptionsmechanismus zur Verwirklichung des autonomen Verhaltens.

Hypothese 4:
Eine kontextfreie Grammatik kann als Beschreibungssprache für Regeln verwendet werden, um flexible und generische Regeln auf intuitive und komfortable Weise auszudrücken und zu formalisieren.

Hypothese 5:
Die Handhabung gemischt kritischer Anwendungen kann durch die Einführung eines Wichtigkeits-Parameters (importance) ermöglicht werden.

Hypothese 1 basiert auf der Annahme, dass ein komplexes System ohne die Hilfe des Systems selbst nicht effizient verwaltet werden kann. Daher erscheint die Integration von autonomem Verhalten durch Selbst-X Eigenschaften als vielversprechender Ansatz zur Bewältigung von MC-CPNs.

Hypothese 2 schlägt ein Middleware-Framework als optimalen Ort vor, um autonomes Verhalten zu implementieren, da die Middleware Zugang zu allen und Wissen über alle Systemkomponenten hat und die notwendigen Abstraktionsebenen für die Anwendungen definiert.

In **Hypothese 3** wird eine MAPE-K-Rückkopplungsschleife mit einer regelbasierten Wissensbasis als Adaptionsmechanismus zur Verwirklichung des autonomen Verhaltens favorisiert, da sie sich bereits im Bereich des autonomen und organischen Computings als wirksam erwiesen hat.

Bei **Hypothese 4** wird von einer kontextfreien Grammatik als effizientem Ansatz zur Formulierung von Regeln ausgegangen, da sie sich durch ihre klare Struktur, Flexibilität und Ausdruckskraft bei der Definition von Bedingungen und Aktionen auszeichnet. Zudem ermöglicht sie eine gute Lesbarkeit und Verständlichkeit der Regeln, was deren

intuitive Formulierung wie auch das Verstehen autonom erlernter Regel-Modifikationen erleichtert.

Abschließend betont **Hypothese 5** die Notwendigkeit der Kenntnis des Kritikalitätsgrads einer Komponente für die Funktionalität des gesamten gemischt-kritischen Systems. Daher wird die Einführung eines Wichtigkeits-Parameters vorgeschlagen. Dieser Parameter ermöglicht eine präzise Beschreibung der Kritikalität und unterstützt somit das effektive Management des Systems.

Die Überprüfung der Validität dieser fünf Hypothesen ist das Hauptziel der vorgelegten Arbeit. Daher präsentiert diese Arbeit eine umfassende adaptive Middleware für MC-CPN (*Chameleon- comprehensive adaptive mixed-criticality supporting middleware for Cyber-Physical Networks*). Der umfassende Ansatz betrachtet alle Ebenen des Systems bestehend aus Anwendungen, Rechenknoten, Kommunikationskanälen, Sensoren, Aktoren und der Middleware selbst. Hierbei werden wesentliche Parameter von Komponenten jeder Ebene wie auch die Struktur und Verteilung selbst angepasst und verwaltet. Daher geht dieser Ansatz klar über individuelle Forschungsarbeiten in den Bereichen Echtzeit-Scheduling, Task-Allokation und Fehlertoleranz hinaus.

***Chameleon* Middleware Architekur** Die entwickelte Middleware *Chameleon* übernimmt das Management und die Verwaltung aller Interaktionen zwischen Anwendungen, Sensoren und Aktoren innerhalb eines einzelnen CPS sowie des gesamten CPNs. Sie ist auch zuständig für die Verwaltung der Ressourcen wie Rechenknoten und Kommunikationskanäle, um einen verteilten Betrieb zu ermöglichen. Aufgrund ihrer zentralen Rolle ist die Middleware der ideale Ort, um eine Selbstanpassung zu ermöglichen, die das gesamte CPN betrifft.

Die entwickelte *Chameleon* Middleware zeichnet sich durch einen modularen Aufbau in Form eines Frameworks aus, das eine einfache Portierung und Modifizierung ihrer Komponenten ermöglicht. Sie kann in zwei Hauptteile unterteilt werden: Zum einen den grundlegenden Middleware-Teil, der für die Bereitstellung verteilter Systeminteraktion und Transparenz verantwortlich ist, und zum anderen die Adaptionslogik, die den Selbstanpassungsprozess steuert.

Der grundlegende Middleware-Teil stellt die notwendigen Funktionen und Mechanismen bereit, um eine effiziente Kommunikation, Koordination und Interaktion zwischen den verschiedenen Komponenten des CPN zu gewährleisten. Er sorgt für die Verbindung und den Austausch von Daten zwischen den Anwendungen, Sensoren und Aktoren, um die gewünschten Aufgaben zu erfüllen.

Die Adaptionslogik ist für den Selbstanpassungsprozess der Middleware verantwortlich. Sie ermöglicht es, dass die Middleware autonom auf Änderungen in der Umgebung, den Anwendungsanforderungen oder den Ressourcen reagiert und entsprechende Anpassungen vornimmt. Dies umfasst die Überwachung des Systemzustands, die Analyse von Daten und Informationen, die Planung von Anpassungsmaßnahmen und die Durchführung der entsprechenden Aktionen. Die Adaptionslogik gewährleistet somit die Flexibilität und Robustheit des CPN, indem sie sicherstellt, dass das System

effizient und zuverlässig agiert, selbst unter sich ändernden Bedingungen.

Insgesamt ermöglicht die entwickelte *Chameleon* Middleware eine effektive Verwaltung und Anpassung von CPNs, indem sie sowohl die grundlegenden Middleware-Funktionen als auch den Selbstanpassungsprozess bereitstellt. Durch ihre modulare Struktur und ihre Fähigkeit, auf Änderungen zu reagieren, bietet sie eine flexible und skalierbare Lösung für das komplexe Management von CPS und CPNs.

Adaptionsmechanismus und *Rango* Regelsprache Dank des gewählten Framework Designs der Middleware können verschiedene Anpassungsmechanismen in die Middleware integriert werden. In dieser Arbeit wurde gemäß Hypothese 3 eine MAPE-K Rückkopplungsschleife verwendet. Diese Rückkopplungsschleife basiert auf einer architektonischen Blaupause, die von IBM im Rahmen des autonomen Computing eingeführt wurde. Sie besteht aus den Komponenten Monitor, Analyze, Plan und Execute, die gemeinsam eine Wissensbasis (Knowledge) teilen.

Eine mögliche Realisierung der Analyze- und Plan-Komponenten sowie der Wissensbasis ist ein Learning Classifier System (LCS). LCS-basierte Ansätze sind im Bereich des Organic Computing beliebt und wurden bereits in einigen CPS-Anwendungen eingesetzt. Im Vergleich zu anderen Online-Lernverfahren zeichnen sich LCS durch einen geringen Rechenaufwand und deterministisches Zeitverhalten aus.

Allerdings bleibt eine wichtige Herausforderung bestehen: die Formulierung des Regelsatzes. Aus diesem Grund wurde in dieser Arbeit die Regelsprache *Rango* entwickelt, die eine generische und flexible Regelformulierung ermöglicht. Üblicherweise werden in LCS Bit-Strings zur Kodierung von Regeln verwendet. Zum Beispiel könnte die Regel "11 -> 1" bedeuten: "Wenn das Auto ein Hindernis vor sich hat und das Auto in Bewegung ist, dann bremse". Äquivalent dazu könnte die Regel "01 -> 0" bedeuten: "Wenn das Auto kein Hindernis vor sich hat und das Auto in Bewegung ist, dann bremse nicht".

Die Verwendung von Bit-Strings zur Repräsentation von Regeln für den hier vorgesehenen Adaptionsmechanismus ist zwar möglich, aber nicht besonders vorteilhaft. Bei komplexeren MC-CPN-Anwendungsfällen würden die Bit-Strings sehr lang werden, was einen erheblichen Aufwand für die Kodierung und eine umständliche Lesbarkeit und Schreibweise der Regeln bedeuten würde. Aus diesem Grund wurde die flexible Sprache *Rango* entwickelt. Sie ermöglicht eine größere Anpassungsfähigkeit, eine intuitive Formulierung der Regeln und berücksichtigt deren dynamische Natur.

Dabei bietet *Rango* eine hohe Flexibilität, um zum einen abstrakte und generische Regeln unabhängig von einem bestimmten Anwendungsfall zu definieren, sowie zum anderen auch anwendungsspezifische Regeln zu erzeugen, wenn dies gewünscht ist.

Generell bietet *Rango* drei wesentliche Vorteile für die Spezifikation des Anpassungsverhaltens in MC-CPNs: Erstens enthält es einen großen Satz vordefinierter Schlüsselwörter für CPN/CPS-Anwendungsfälle, die wiederverwendet werden können. Zweitens ist *Rango* direkt in die Lernkomponente integriert. Die mit *Rango* spezifizierten Regeln werden automatisch in eine binäre Repräsentation übertragen, die von einem LCS als Grundlage für den Lernprozess verwendet werden kann. Drittens

ermöglicht dieser Ansatz erklärbar künstliche Intelligenz (XAI), da er die Nachvollziehbarkeit und Erklärbarkeit des Lernens verbessert. Es besteht die Möglichkeit, einen durch das Lernen modifizierten Regelsatz in einer menschenlesbaren Form von zu exportieren.

Implementation Die *Chameleon* Middleware wurde in C++ implementiert und zur praktischen Evaluierung in eine simulierte Umgebung eingebettet, die auf OMNet++ basiert. OMNet++ ist ein bekannter und weit verbreiteter ereignisbasierter Simulator für die Netzwerkverarbeitung. Er bietet eine Vielzahl von Funktionen und vordefinierten Modulen zur präzisen Simulation von Kommunikationsnetzwerken sowie Sensor- und Aktorsystemen.

Für die spezifische Evaluierung wurden in OMNet++ zusätzliche Module zur Simulation realistischer Anwendungen und Laufzeitdynamik integriert. Darüber hinaus wurden vorhandene OMNet++-Module angepasst, um Rechenknoten zu simulieren. Auf diese Weise bietet die simulierte Umgebung eine grundlegende Plattform für die Middleware-Forschung im Bereich gemischt-kritischer CPS und CPNs. Sie ermöglicht die effiziente Ermittlung, Implementierung, Kalibrierung und Evaluierung von Anwendungsanforderungen in solchen Systemen.

Der gewählte Ansatz für die Simulation ist hierbei monolithisch, was Vorteile in Bezug auf Effizienz und Zeitverhalten im Vergleich zu Co-Simulatoren bietet. Insbesondere für die Simulation von Echtzeitumgebungen, die für MC-CPNs von großer Bedeutung sind, ist der monolithische Ansatz vorteilhaft. Bei Co-Simulatoren ist die Koordination zwischen den einzelnen Simulatoren sehr komplex und führt häufig zu Latenzproblemen im System.

Im Vergleich zu einer realen Welt bietet die simulierte Umgebung den Vorteil, einen tiefen Einblick in die Anpassungsprozesse zu ermöglichen, und physikalische Schäden bei Defiziten zu vermeiden. Außerdem ermöglicht sie eine größere Flexibilität hinsichtlich verschiedener Anwendungsszenarien und der Größe der simulierten Umgebung. Es ist jedoch wichtig zu betonen, dass die *Chameleon* Middleware vollständig unabhängig von der simulierten Umgebung ist und mit minimalen Anpassungen der Schnittstellen in einer realen Umgebung eingesetzt und getestet werden kann.

Evaluation Die Evaluation von *Chameleon* umfasst zahlreiche Aspekte, um die Gültigkeit der aufgestellten Forschungshypothesen zu überprüfen. Im folgenden sind die wesentlichen Evaluationsaspekte zusammengefasst, die zur Beurteilung von *Chameleon* herangezogen wurden:

1. Nützlichkeit, Benutzerfreundlichkeit und Leistungsfähigkeit der Regelsprache *Rango*: Die Regelsprache *Rango* wurde zum einen hinsichtlich ihrer Nützlichkeit und Benutzerfreundlichkeit bewertet. Es wurde untersucht, wie gut sie sich zur Spezifikation von Anpassungsverhalten eignet, inwieweit gelerntes Verhalten nachvollzogen werden kann und wie einfach sie von den Entwicklern verwendet werden kann. Darüber hinaus wurde die Leistungsfähigkeit der Regelsprache in

Bezug auf den Speicherbedarf für Regeln und die Komplexität des Parsings und der Regelauswertung zur Laufzeit evaluiert.

2. Echtzeitverhalten von *Chameleon*: Das Echtzeitverhalten von *Chameleon* hinsichtlich zweier Aspekte untersucht. Zum einen wurde das Echtzeitverhalten des Adaptionsmechanismus theoretisch analysiert und garantierte Oberschranken für die Dauer eines Adaptionsprozesses abgeleitet. Zum anderen wurde das Echtzeitverhalten der Anwendungsabläufe betrachtet, welches basierend auf gegebenen Echtzeitanforderungen sowie des *importance* Wertes bei MC-CPNs vom Adaptionsmechanismus überwacht und gesteuert wird.

3. Effizienz und Effektivität des regelbasierten MAPE-K Adaptionssmechanismus: In einem automobilen Anwendungsszenario wurde die Funktionsfähigkeit und Wirksamkeit des regelbasierten MAPE-K Adaptionssmechanismus zur Verwaltung von MC-CPNs und deren Komplexität unter Berücksichtigung der gemischten Kritikalität untersucht. Hierfür wurde die OMNet++ basierte Simulationsumgebung verwendet. Es wurden verschiedene Evaluationsszenarien benutzt, um den Umgang mit dynamischen Laständerungen, Überlastsituationen, Ausfällen, Designfehlern, Lerneffekten und der Nutzung verfügbarer Ressourcen zu bewerten. Zusätzlich bestätigten die Evaluationscenarien die hergeleiteten Echtzeitgrenzen des Adaptionsmechanismus.

Fazit Die umfassende Evaluierung zeigt zum einen, dass die entwickelte *Chameleon* Middleware das autonome, effiziente, effektive, zuverlässige und flexible Management von MC-CPNs und ihrer Komplexität unter Berücksichtigung der gemischten Kritikalität ermöglicht. Zum anderen belegt sie die Gültigkeit der fünf aufgestellten Forschungshypothesen.

Die Realisierung von autonomem Verhalten durch Selbst-X Eigenschaften hat sich als erfolgreicher Ansatz erwiesen, um das Management und die Komplexität von MC-CPNs zu bewältigen (*Hypothese 1*). Durch die Kapselung dieses autonomen Verhaltens in einer Middleware-Schicht kann das MC-CPN sich selbst organisieren und an veränderliche Bedingungen und Systemziele anpassen, ohne zentrale Steuerung oder menschliche Interaktion zu erfordern (*Hypothese 2*). Dabei berücksichtigt der umfassende Ansatz das gesamte System, bestehend aus Anwendungen, Rechenknoten, Kommunikationskanälen, Sensoren, Aktoren und der Middleware selbst. Die Fähigkeiten von *Chameleon* gehen somit über die spezifische verwandte Forschung hinaus, da sie eine Vielzahl von Aspekten adressiert und den Systementwicklern die Last der Verwaltung komplexer MC-CPNs abnimmt.

Weiterhin wurde festgestellt, dass eine regelbasierte MAPE-K-Rückkopplungsschleife ein praktikabler Anpassungsmechanismus ist, um Self-X Eigenschaften in der Middleware umzusetzen (*Hypothese 3*). Die Verwendung eines verteilten LCS zur Instanziierung der MAPE-K Rückkopplungsschleife ermöglicht echtzeitfähige Adaptionsmaßnahmen und ist durch seinen geringen Rechenaufwand auch in ressourcenbeschränkten Systemen einsetzbar. Zudem hat sich gezeigt, dass *Rango* einen intuitiven Weg bi-

etet, um einen anfänglichen Regelsatz für das LCS im Kontext von MC-CPNs zu spezifizieren und die Systemadministratoren und Entwickler bei der Generierung des Regelsatzes zu unterstützen (*Hypothese 4*). Aufgrund der verteilten Natur von MC-CPNs verwendet *Chameleon* eine verteilte Regelauswertung ohne zentrale Steuerung. Hierbei finden zusätzliche Strategien zur Behandlung von Konflikten Anwendung. Die Regelsprache ermöglicht eine flexible Kombination verschiedener Anpassungsaktionen und -bedingungen in Bezug auf gemischte Echtzeitanforderungen von Anwendungen. Sie ermöglicht auch die Kombination verschiedener Geltungsbereiche für Bedingungen und Aktionen, um das Systemverhalten zu ändern. Hierbei hat sich schließlich die Einführung des Wichtigkeits-Parameters (*importance*) als wirkungsvolle Methode zur Behandlung gemischter Kritikalität erwiesen. (*Hypothese 5*).

Zusammenfassend trägt *Chameleon* durch die Einführung und Kombination der folgenden Konzepte zum aktuellen Forschungsstand bei:

- Ein umfassender Selbstanpassungsmechanismus auf allen Ebenen des Systemmodells wird bereitgestellt.
- Dieser Mechanismus erlaubt eine flexible Kombination von parametrischen und strukturellen Anpassungsmaßnahmen (Relokalisierung, Scheduling, Tuning, ...), um das Verhalten des Systems zu verändern.
- Echtzeitbeschränkungen von gemischt-kritischen Anwendungen (harte Echtzeit, weiche Echtzeit, Best-Effort) werden in allen möglichen Anpassungsbedingungen und -aktionen durch die Verwendung des Parameters *importance* berücksichtigt.
- CPNs werden durch die Einführung verschiedener Geltungsbereiche (lokal, System, global) für die Anpassungsbedingungen und -aktionen unterstützt. Dies ermöglicht auch die Kombination von unterschiedlichen Geltungsbereichen für Bedingungen und Aktionen.
- Die Instanziierung der MAPE-K Rückkopplungsschleife durch ein verteiltes LCS ermöglicht eine echtzeitfähige Schlussfolgerung von Anpassungsaktionen, welche auch auf ressourcenarmen Systemen funktioniert.
- *Rango* bietet eine intuitive Möglichkeit, einen initialen Regelsatz für LCS im Kontext von CPS/CPNs zu spezifizieren und unterstützt die Systemadministratoren bei der Regelsatzgenerierung.

Schlüsselworte— Middleware, Cyber-Physical Networks, Cyber-Physical Systems, Gemischte Kritikalität, Selbst-X Eigenschaften, (autonome) Adaption

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

The rapid growth of Cyber-Physical Systems (CPS) due to the availability of cheap hardware, sensors, actuators and communication links, has led to an increased system complexity. In the automotive area e.g. the number and capabilities of these components in a car is constantly rising during the last years. This complexity arises due to the interdependence of physical processes, sensing and actuation, communication networks, and computing resources. The combination of these factors creates a multitude of issues that must be addressed, such as task allocation, scheduling, fault tolerance and communication. As the number of heterogeneous components in such a system rises, the complexity of the interactions between them also increases. A network of cooperating CPSs - a so-called Cyber Physical Network (CPN) - makes the situation even more complex, since information between the CPSs in the network has to be exchanged and actions have to coordinated and deployed. Cooperating or platooning vehicles [Les+21], in which several autonomous vehicles drive in a convoy with small inter-vehicle distances and coordinate their inter-vehicle gaps via IEEE 802.11p communication, can be an example for such a CPN.

Additionally, CPNs are often deployed in dynamic, unpredictable environments and safety-critical domains, such as transportation, energy, and healthcare. In such domains, usually applications of different criticality level exist. In an automotive environment for example, the brake has a higher criticality level regarding safety as the infotainment. As a result of mixed-criticality, applications requiring hard real-time guarantees compete with those requiring soft real-time guarantees and best-effort application for the given resources within the overall system. This leads to the need to accommodate multiple levels of criticality while ensuring safety and reliability, which increases the already high complexity even more. This high level of complexity has made it difficult to manage and optimize the performance, reliability, and safety of such mixed-critical CPNs (MC-CPN). Thus, it is essential to develop new approaches and techniques to ensure that MC-CPNs can operate under a range of conditions while meeting stringent requirements.

In summary, the motivation for research in the area of MC-CPNs is driven by the need to develop efficient, reliable, and scalable solutions for managing the complexity and heterogeneity of these systems while ensuring safety.

Autonomous behavior refers to a set of properties that are capable to operate and control a system in a self-organizing way (cf. Section 2.3.1). These properties, which are usually subsumed as so-called self-X properties, are a leverage for MC-CPNs since the increasing complexity cannot be handled effectively and efficiently by the system

developer without the assistance of the system itself any longer [HM08]. Therefore, the development of new autonomous mechanisms and techniques for managing MC-CPNs looks like a promising approach to improve their performance, efficiency, reliability and flexibility.

1.2. Approach

This thesis deals with the question on how to conveniently, effectively and efficiently handle the management and complexity of MC-CPN's. To answer this question, several research hypotheses are formulated (cf. Figure 1.1):

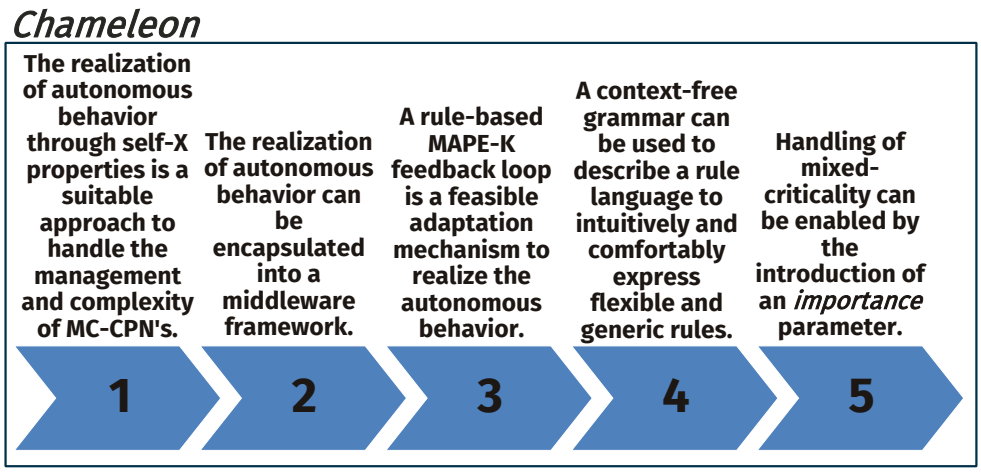


Figure 1.1: Research hypotheses for the design and development of *Chameleon*.

Hypothesis 1:
The realization of autonomous behavior through self-X properties (e.g. self-adaptation, self-organization, ...) is a suitable approach to handle the management and complexity of MC-CPN's.

Hypothesis 2:
The realization of autonomous behavior can be encapsulated into a middleware framework.

Hypothesis 3:
A rule-based MAPE-K feedback loopis a feasible adaptation mechanism to realize the autonomous behavior.

Hypothesis 4:
A context-free grammar can be used to describe a rule language to intuitively and comfortably express flexible and generic rules.

Hypothesis 5:

Handling of mixed-criticality can be enabled by the introduction of an importance parameter.

Hypothesis 1 is based on the assumption that a complex system cannot be efficiently managed without the assistance of the system itself. Therefore, gaining autonomy by realizing self-X properties looks like a promising way to manage MC-CPNs.

Hypothesis 2 proposes a middleware framework as best suitable place to realize the autonomous behavior, because the middleware has access to and knowledge about all system components and allows to define the necessary layers of abstraction for the application.

In **Hypothesis 3**, a *M*onitor-*A*nalyze-*P*lan-*E*xecute (*MAPE*) loop with rule-based *K*nowledge (K) is favored as adaptation mechanism to realize the autonomous behavior, because it has already shown its effectiveness for that purpose in Autonomic and Organic Computing (cf. Section 2.3).

Thereby, **Hypothesis 4** assumes a context-free grammar as an efficient method to express the rules, because such a grammar is well-structured, flexible and powerful in expressing conditions and actions for rules. Additionally, it offers a good writeability and readability which is advantageous for writing rules as well as understanding modifications achieved by autonomous learning. Finally, the criticality level of a component for the functionality of the entire mixed-critical system has to be known by the system.

Therefore, **Hypothesis 5** proposes the introduction of an *importance* value to express this level. To examine the validity of these five hypotheses is the major goal of the presented thesis. Therefore, this thesis introduces a *comprehensive adaptive mixed-criticality supporting middleware for Cyber-Physical Networks* (*Chameleon*), which efficiently and autonomously handles the management and complexity of CPN with regard to the mixed-criticality aspect. The entire system consisting of applications, computing nodes, communication channels, sensors, actuators and the middleware itself is considered. In such a comprehensive approach, the key parameters of each of these components (e.g. scheduling parameters, scheduling schemes, task parameters, communication parameters and protocols, data compression, monitoring rates, ...) as well as the structure itself (task allocation) are subject of adaptation and management. Thus, the work presented here addresses a multitude of issues simultaneously and clearly exceeds individual research in the areas of classical real-time scheduling, task allocation and fault tolerance.

1.3. Structure

The thesis is organized as follows: First, the most important fundamentals relevant for this thesis are introduced in Section 2. Following in Section 3, the system model which describes the world in which *Chameleon* middleware operates is shown. The architecture of *Chameleon* is presented in Section 4 and the rule based MAPKE-K adaptation mechanism is explained in Section 5. Afterwards, the implementation of *Chameleon* is

introduced in Section 6. An extensive evaluation is presented and discussed in Section 7, followed by related work in Section 8. Finally, Section 9 concludes this thesis with a discussion of the results regarding the five research hyptheses, a summary and an outlook.

2. Fundamentals

In the following, important concepts relevant for this thesis are introduced.

2.1. Embedded System, Cyber-Physical System and Cyber-Physical Network

Embedded Systems (ES), Cyber-Physical Systems (CPS), and Cyber-Physical Networks (CPNs) are interrelated concepts that are used to describe different levels of integration between physical systems and computing systems. Together, these concepts enable the seamless integration of physical and computing systems, improving the development of advanced and intelligent systems. The individual concepts are described in more detail in the following.

2.1.1. Embedded System

An embedded system (ES) [Noe12] as shown in Figure 2.1 is a system that interacts with and controls the environment via sensors and actuators. Thus, it consists of a combination of hardware and software. It is often designed to perform tasks with real-time constraints and is usually integrated into a larger system. Embedded systems are an important part of the modern world, and are used in a wide variety of applications. Examples for ESs can be found e.g. in the automotive area, where a vehicle has a variety of ESs responsible for controlling various functions such as engine control, brakes, infotainment system, etc.

2.1.2. Cyber-Physical System

A CPS is a system which consists of several subsystems (system-of-systems) which interact and cooperate via a communication system (cf. Figure 2.1). Thereby, at least one of the subsystems is an ES which interacts with the environment [Mar21]. Thus, ESs are the building blocks of a CPS. Thereby, the CPS relies on ESs to monitor the physical environment via sensors and make decisions based on the data collected. The CPS provides the computational intelligence needed to control the environment via actuators in real-time. This integration enables a wide range of applications. A vehicle itself is a good example for a CPS, since — as mentioned above — it holds several ESs which cooperate to enable the overall vehicle functionality.

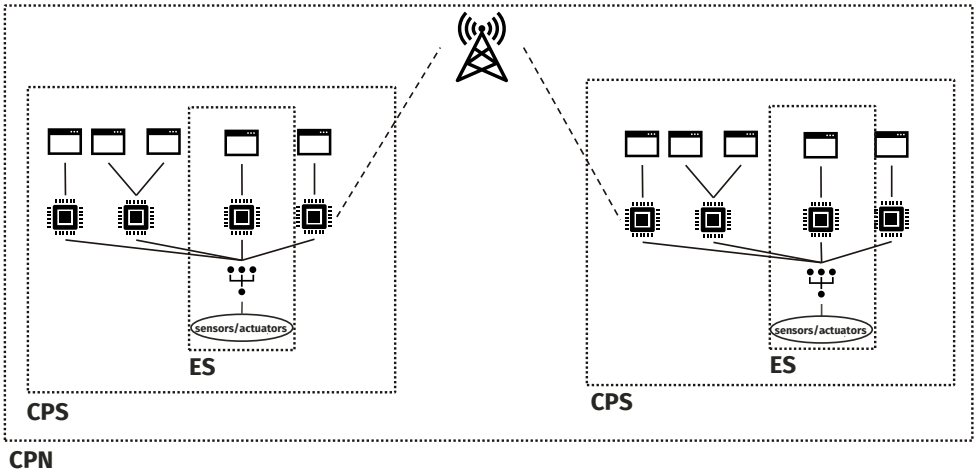


Figure 2.1: A Cyber-Physical Network (CPN) consisting of multiple Cyber-Phyiscal Systems (CPS). Thereby, in each CPS at least one of the subsystems is an Embedded System (ES) which monitors and controls the environment via sensors and actuators.

2.1.3. Cyber-Phyiscal Network

A CPN connects multiple CPSs together to form a network of interconnected systems (system-of-systems in a larger scale). These networks enable communication and coordination between multiple CPS, allowing for more complex and sophisticated control and optimization of physical systems. The relationship between ES, CPS and CPN is illustrated in Figure 2.1. Platooning vehicles are a good example for a CPNs, where each vehicle represents a CPS and consists of several ESs.

2.1.4. Application Fields

CPNs are used in a wide variety of applications to increase efficiency, reduce energy consumption and costs and enhance safety. Some examples are provided in the following:

- Transportation and Logistics: As already mentioned above, CPNs can be used to organize sets of vehicles, e.g. for platooning, as well as a vehicle itself. Thus, CPN's can be used to monitor and control traffic flow, optimize routes, and reduce congestion.
- Smart Homes and Building: CPN's can be used to automate and optimize various functions in homes and buildings, such as lighting, heating, ventilation, and security.
- Healthcare: CPN's can be used to monitor and manage patient health, optimize treatment plans, improve diagnostics, deliver drugs and other treatments and

automate medical procedures.

- Manufacturing and Industrial Automation: CPNs can be used to automate and optimize manufacturing processes by integrating sensors, actuators and control systems.
- Smart Grid and Energy Management: CPNs can be used to monitor and control energy distribution networks, optimize energy usage by enabling real-time optimization, and integrate renewable energy sources.
- ...

Chameleon can be used in any of the presented applications fields by choosing appropriate parameters and rules according to the individial requirements and goals.

2.2. Mixed-Critical Systems

Mixed-critical systems refer to systems that handle multiple tasks of different criticality levels, such as safety-critical and non-safety critical tasks, on the same hardware platform (or within the same system). The distinction into different criticality levels can be much more fine-grained than only safety-critical and non-safety critical tasks depending on the use case. In the automotive area for example, *ASIL* (*A*utomotive *S*afety *I*ntegrity *L*evel) [al18] distinguishes safety integrity levels between *A* and *D*. Thus, mixed-critical systems require to support multiple applications with different criticality levels, where each application may have different requirements in terms of performance, fault-tolerance and safety. As a result, these systems require a careful design to ensure that the criticality of the tasks is taken into consideration during the system runtime. For example, safety-critical tasks should not by compromised by the non-safety critical ones, so that the system as a whole meets the required safety standards.

2.3. Autonomic and Organic Computing

Autonomic and Organic Computing are two related research fields in computer science that aim to create more flexible, adaptable and self-organizing computer systems.

In 2002, the Architecture of Computer Science (ARCS) specialist group of the *Gesellschaft für Informatik* (GI) held a workshop aimed at predicting future trends in computer science [TSM17; MT17; Sch05]. From this workshop emerged the concept of Organic Computing, which was later published in a joint position paper by the GI and the *Informationstechnischen Gesellschaft im VDE* (ITG) in 2003 [VDE03]. Organic Computing involves creating computer systems that can adapt and evolve like living organisms, with the ability to sense and respond to their environment, learn from experience, and adapt to changing circumstances. Such systems are designed

to be robust and resilient, capable of recovering from failures and maintaining their functionality in the face of changing conditions.

In 2003, Autonomic Computing emerged as a concept around the same time as Organic Computing. Initially proposed by IBM, the primary goal was to develop automated solutions for data centers [KC03]. Autonomic Computing emphasizes self-management and draws inspiration from nervous systems. Its focus is on creating computer systems that can monitor their own performance, diagnose and repair faults, and adjust their behavior in response to changing conditions, without the need for human intervention. By doing so, Autonomic Computing aims to create more efficient and reliable systems that can deliver high levels of performance and availability with minimal manual tuning and configuration.

Both Organic and Autonomic Computing are motivated by the increasing complexity and scale of modern computer systems, which can be difficult and time-consuming to manage manually. By creating systems that are more flexible, adaptable, and self-organizing, it is possible to create computer systems that can better meet the needs of users and organizations in a wide range of domains, from scientific research to business and industry.

2.3.1. Self-X Properties

In the context of Autonomic and Organic Computing, self-X properties refer to a set of properties that are exhibited by complex systems that are capable of self-organization, self-adaptation, and other forms of autonomous behavior. The term "X" is used as a placeholder to represent the specific property being discussed, such as self-organization, self-adaptation, and so on.

Major self-X properties include:

- **Self-organization**: The ability of a system to spontaneously organize and structure itself in response its environment and internal state. It is a kind of umbrella term for the other self-X properties.
- **Self-adaptation**: The ability of a system to adapt its behavior or structure in response to changes in its environment or internal state, based on some predefined criteria or goal.
- **Self-monitoring**: The ability of a system to monitor its own behavior and internal state, and adjust its behavior accordingly.
- **Self-optimization**: The ability of a system to optimize its performance based on feedback from its environment and performance metrics.
- **Self-configuration**: The ability of a system to autonomously find an initial working configuration.
- **Self-healing**: The ability of a system to react to and heal failures occurring during system operation.

When applied to a specific use case like platooning, the self-X properties can greatly enhance both the efficiency and safety of the platoon while minimizing disruption to traffic. One example is self-organization, which can be achieved through the use of sensors and communication systems to determine the optimal formation, speed, and spacing between the vehicles. This can help reduce wind resistance, enhance fuel efficiency, and minimize the risk of accidents. Furthermore, platooning vehicles can self-adapt to changing road and traffic conditions. For instance, if there is an obstacle on the road, the platoon can self-adjust its position and speed accordingly to avoid accidents and maintain smooth traffic flow. In addition, platooning vehicles can self-heal by detecting and responding to faults and failures in the system. If one vehicle experiences a mechanical issue, the other vehicles can adjust their speed and position to maintain the platoon formation, ensuring smooth traffic flow. In case of computational issues (e.g. failures or overloads in computing nodes or networks) the resources of other vehicles in the platoon can be used for self-healing.

Overall, self-X properties are important characteristics of complex systems that exhibit autonomous behavior, and are often desired in many application domains, including CPSs, distributed systems, and artificial intelligence. These properties allow the system to autonomously adapt and respond to changing conditions, and optimize their behavior to achieve their objectives, while providing a high degree of reliability and efficiency with only minimal or no need at all for external intervention or control.

In the context of CPS/CPN, self-X properties refer to the ability to exhibit self-organization mainly in terms of self-adaptation and self-healing to keep the system operational within the given mixed-critical constraints, which is important for achieving efficient and reliable system operation in dynamic and unpredictable environments.

In the field of Organic Computing, often the Observer-Controller control mechanism is used to realize self-X properties while in Autonomic Computing the MAPE-K feedback loop is common. Both concepts are shortly introduced in the following.

2.3.2. Observer-Controller Architecture

The Observer-Controller architecture [Hel+04] (see Figure 2.2) is a control mechanism used to regulate the behavior of a system by observing and responding to changes in the system and its environment.

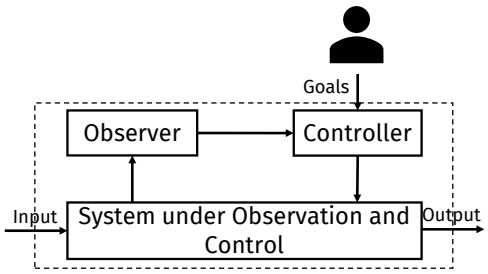


Figure 2.2: Observer-Controller archiecture.

In this architecture, the system under observation and control is extended by two main components: the *observer* and the *controller*. The *observer* continuously monitors the system's behavior and the environment in which it operates, collecting data and generating models of the system's state and dynamics. The *controller* receives this information from the *observer* and generates control signals that modify the system's behavior in response to changes in the environment or to achieve specific goals.

Overall, the Observer-Controller architecture is a flexible and scalable mechanism that can be used to regulate the behavior of complex and dynamic systems in an autonomous and adaptive manner.

2.3.3. MAPE-K Feedback Loop

The MAPE-K feedback loop [IBM05] (see Figure 2.3) is a control paradigm to regulate the behavior of a system in an autonomous and adaptive manner. The loop consists of four main components:

- **Monitor (M)**: The monitor component continuously observes the system's behavior and environment, collecting data of the system's state and dynamics.
- **Analyze (A)**: The analyze component receives the data and analyzes it to identify potential problems or opportunities for improvement.
- **Plan (P)**: The plan component generates a set of actions or policies based on the analysis performed by the analyze component.
- **Execute (E)**: The execute component carries out the actions or policies generated by the plan component, modifying the system's behavior in response to changes in the environment or to achieve specific goals.

Often those components share a **Knowledge (K)** base.

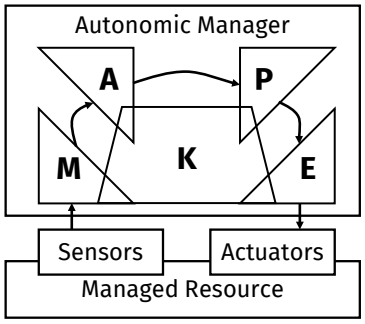


Figure 2.3: MAPE-K feedback loop archiecture.

The closed-loop nature of the MAPE-K loop enables the system to continuously adapt and improve its behavior based on the changing environment and system requirements

Overall, the MAPE-K feedback loop provides a flexible and scalable mechanism for regulating the behavior of complex and dynamic systems in an autonomous and adaptive manner.

In general, MAPE-K and Observer/Controller have many similarities and often use the same mechanisms, e.g. rule-based learning mechanisms. While MAPE-K is more a generic design pattern, Observer/Controller refers more to a concrete architecture.

2.3.4. Learning Classifier Systems

A Learning Classifier System (LCS) is a powerful and flexible approach to machine learning that is inspired by the human immune system and evolutionary theory. LCSs were first proposed by John Holland in the 1970s as a way of modeling the adaptive behavior of natural systems [Hol+00; SW09; UM09; UB17] and can be used in the Observer-Controller architecture as well as in the MAPE-K feedback loop. One of the key features of LCS is its ability to learn and adapt over time. This makes LCS well-suited for applications where the ability to learn from experience and adapt to changing conditions is crucial.

A LCS consists of a set of rules, or classifiers, that are used to make predictions or decisions based on input data. Such classifiers are usually represented as binary strings and consists of an condition part, an action part, and a reward for executing the action. The workflow of a LCS is outlined in Figure 2.4. The LCS uses sensors to monitor the

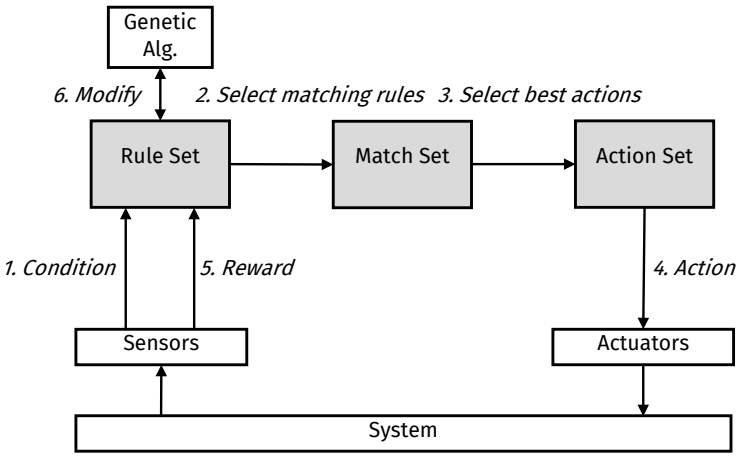


Figure 2.4: Workflow of a LCS.

system and to retrieve the current system condition. The conditions of the classifiers are compared to the current condition and classifiers whose conditions fit are included to the *match set*. Afterwards, the best rewarded actions of the classifiers in the *match set* are written to the *action set* and are executed using the actuators. Subsequently, the system condition is evaluated again and the reward of applied classifiers is updated according to the evaluation before the LCS loop is repeated.

Often, these classifiers are generated and modified using a process called *genetic algorithm*, which is a type of optimization algorithm inspired by natural selection. The genetic algorithm works by randomly generating an initial population of classifiers, evaluating their performance on a given task, and then selecting the best-performing classifiers for reproduction. The selected classifiers are then combined through crossover and mutation to create a new generation of classifiers, which are evaluated and selected again. However, the random generation of classifiers might not lead to an appropriate rule set in the end. Instead, expert-knowledge can be used to build an initial rule set.

2.4. Middleware

The term middleware describes a software layer used in distributed systems to simplify the management and development of such systems by providing transparencies which hide the systems complexity [Cou+11; TS01]. In general, such a middleware can be used in various context providing a variety of different transparencies. It spans a common system-wide platform within the distributed system as shown in Figure 2.5 and thereby eases the interaction of the system components.

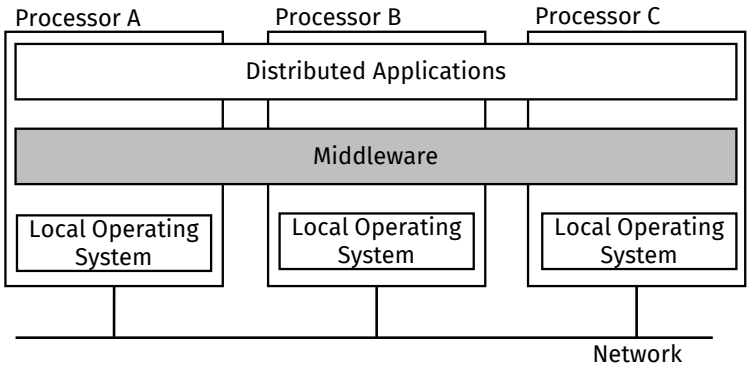


Figure 2.5: Middleware software layer in a distributed system.

In this thesis, the middleware software layer additionally is used to enable autonomous behavior. Thus, the layer provides features that allow the CPS/CPN to self-organize and adapt to changing conditions and system goals without requiring a central control or human interacting. Therefore, the middleware software layer can be used to encapsulate the realization of self-X properties as introduced in Section 2.3.1.

Such a middleware which enables autonomous behavior normally needs to provide the following features:

- **Autonomy**: Each node in the system has a high degree of autonomy and can make decisions based on local and global information without requiring a central authority.
- **Communication**: The middleware provides a communication infrastructure that enables system components to exchange information and coordinate their

actions.

- **Adaptation**: The middleware provides mechanisms to adapt to changes in the environment and system objectives.
- **Emergence**: Emergence is a phenomenon in which a complex system exhibits properties or behaviors that arise from the interactions between its individual components, rather than being directly caused by those components themselves. Thus, complex and often unpredictable behavior can emerge from simple interactions between individual components of a system. An example for emergence is the behavior of a colony of ants. Individual ants deposit pheromones to mark their paths and following the pheromone trails left by other ants. When these individual behaviors are combined, the colony as a whole exhibits emergent properties and behaviors, such as the ability to find the shortest path to food sources.

  The middleware supports and controls the emergence of complex behavior from the interactions of the components in the system.

Overall, a middleware is an essential part to handle the complexity of CPS/CPNs. A middleware for autonomous behavior enables a CPS/CPN to operate in a distributed and decentralized manner, adapt to changing conditions, and handle constraints to achieve its objectives.

3. System Model

The system model shown in Figure 3.1 describes the world in which the *Chameleon* middleware (MW) operates. Due to the comprehensive approach and the required

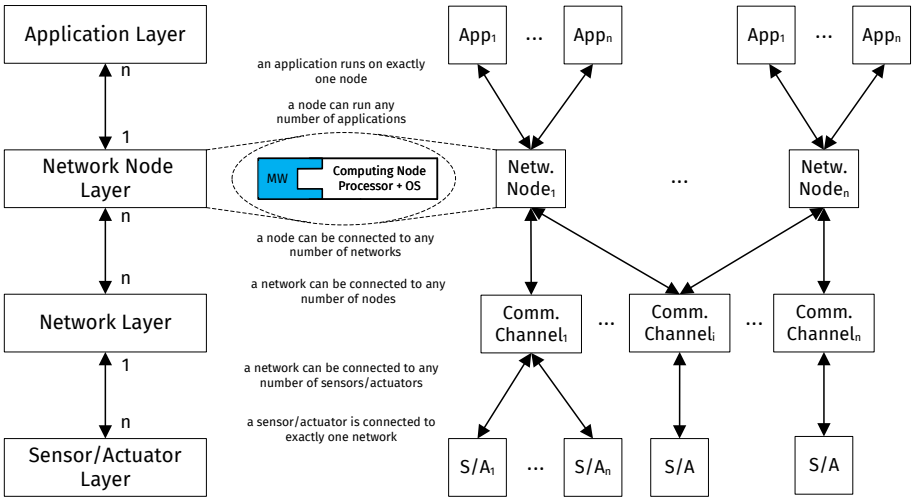


Figure 3.1: System model.

end-to-end view, all the resources of a MC-CPN and their interactions with each other are of concern to the middleware. The application layer represents the application components of the cyber-physical system network. The network node layer contains the computing nodes (processors including the operations system) and the *Chameleon* middleware instances. The network layer is responsible for the interconnection of the nodes and the sensors and actuators in the system. Sensors and actuators reside in the sensor/actuator layer. In the following, further information on the model, its layers and properties is provided. An overview of the major properties of the layers is provided in Table 3.1.

Layer	Property	Remark
Sensor Actuator	importance	The importance of the component
	priority	The priority of the component
	period	The period of a periodic sensor
	deadline	The deadline for sensor data delivery
	reaction time	The reaction time to sensor data available
	data size	The size of data a sensor produces
	jitter	The period jitter of a periodic sensor

	delay	The delay of a reactive sensor
	compression	The current data compression factor
	max compression	The maximum data compression factor
	max miss rates	The maximum allowed miss rates for period, deadline and reaction time
	max distance	The maximum allowed distances to period, deadline and reaction time
Network	capacity	The data transmission capacity (data rate) of the channel
	delay	The data transmission delay
	mode	The data transmission mode (simplex, half duplex, duplex)
	scheduling policy	The available and current scheduling policies
	error rate	The data transmission error rate
	repetition	The repetition scheme in case of error
	channels	The number of parallel transmission channels (lanes) of the communication channel
	failed channels	The number of failed transmission channels (lanes) in a communication channel
Network Node	capacity	The processing capacity (instruction rate) of the node
	delay	The processing delay of the node
	scheduling policy	The available and current scheduling policies
	channels	The number of parallel processing channels (threads) of the node (single or multi-threaded nodes)
	failed channels	The number of failed processing channels (threads) in a multi-threaded node.
Application	importance	The importance of the application
	priority	The priority of the application
	period	The period of a periodic application
	deadline	The deadline of an application
	reaction time	The reaction time of an application
	instruction count	The instruction count of an application
	jitter	The period jitter of a periodic application
	tuning factors	The current tuning factors for priority, period deadline and reaction time
	max tuning	the maximum tuning factor
	compression	The current data compression factor
	max compression	The maximum data compression factor
	max miss rates	The maximum allowed miss rates for period, deadline and reaction time

	max distance	The maximum allowed distances to period, deadline and reaction time

Table 3.1: Major properties of the system model layers.

3.1. Modeling Mixed-Criticality – Importance

One of the most important features of the system model is the notion of mixed-criticality. It means that the different components or tasks in the system might have a different criticality regarding the system functionality and well-being. In an automotive environment for example, the brake component is more critical than the navigation component. Therefore, a parameter called *importance* is defined:

Definition (Importance $\in \mathbb{N}_0$)**.** Level of criticality of a task in the system. The higher the level of *importance* the more critical is the task regarding the system functionality and well-beeing.

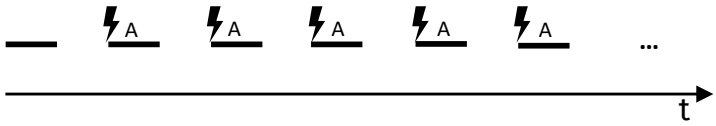
Importance must not be confused with priority.

Definition (Priority $\in \mathbb{N}_0$)**.** Priority refers to the order in which tasks are scheduled based on their period or deadline.

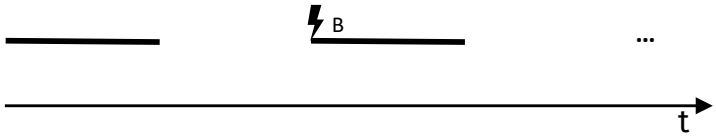
Regarding real-time scheduling (e.g. rate monotonic scheduling), application components with short periods result in high priorities (e.g. infotainment high definition audio rendering). However, these application components might be less important for the system than application components with longer periods and resulting lower priority (e.g. brake light control). In case of overload or lack of resources the lower priority application component then has to be preferred.

An example is illustrated in Figure 3.2. In the example, two tasks with different periods/corresponding deadline (assuming the deadline equals the period; illustrated as thunderbolt) compete for the given resources. Thereby, $Task_A$ has a short execution time with short period/deadline (cf. Figure 3.2a) and $Task_B$ has a longer execution time with longer period/deadline (cf. Figure 3.2b). Thus, the priority of $Task_A$ must be higher than that of $Task_B$ to get a working scheduling using rate monotonic scheduling with preemption (cf. Figure 3.2c). However, it may be the case, that $Task_B$ is the more important of the two, although it has the longer period and therefore the lower priority. If the system now gets into an overload situation (e.g. by loss of computational power or a new task entering the system) the more important task would be neglected due to its lower priority in a pure priority based scheduling. This is where the *importance* comes into play as an additional scheduling decision parameter[1].

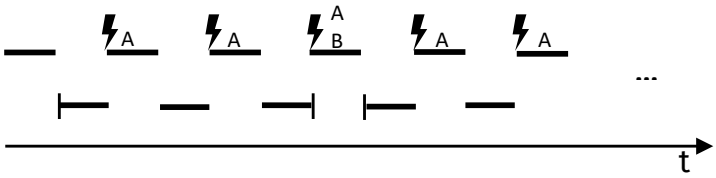
[1] Simply combining both parameters in a two-level scheduling with classification into importance classes and priority scheduling based on the classes does not work. If $Task_B$ is classified in importance class 1 and $Task_A$ in importance class 2, $Task_B$ always has priority over $Task_A$. This means that $Task_A$ could never meet its deadline in the above example.



(a) $Task_A$: Short execution time with short period/deadline.



(b) $Task_B$: Longer execution time with longer period/deadline.



(c) Rate monotonic scheduling with preemption.

Figure 3.2: Example: Priority scheduling.

By introducing a cardinal valued *importance* parameter, the criticality of a component or task can be expressed on a fine-grained level independent of its priority.

3.2. Sensor/Actuator Layer

In the system model, a sensor can either be a periodic sensor that sends data periodically, a reactive sensor that only sends data if requested beforehand, or a sporadic sensor that sends data sporadically. For each sensor, several real-time properties (periods for periodic sensors, priorities, deadlines,...) can be defined. Also, the *importance* parameter, which defines the importance of the sensor data for the system, and thus expresses its criticality, can be defined. An actuator is modeled as a data endpoint having additional real-time tolerance parameters like the maximum allowed miss rate of its timing constraints and the maximum allowed distance to its timing constraints.

3.3. Network Layer

The communication channels[1] of the network layer are responsible for the communication between the different parts of the MC-CPN by offering the opportunity to transmit

[1] A communication channel is the physical medium including the protocol used for transmission: E.g. TCP/IP and UDP/IP are two different communication channels even when they are built upon the same physical medium like Ethernet.

messages through it. A channel might be a peer-to-peer connection, a databus or an n-to-m connection network (wired or wireless). Thereby, a channel can support simplex, half-duplex or full duplex transmission mode and can have multiple parallel transmission lanes. Its data transmission capacity (data rate) as well as its transportation delay can be defined. Furthermore, a channel can support various scheduling policies (e.g. FIFO or priority scheduling, preemption or no preemption). Error-rates, repetition schemes and parallel transmission channel (lane) failures in multi-lane communication channels can also be taken into account.

3.4. Network Node Layer

Besides the delay due to communication, the delay due to processing (application & middleware) has also to be considered. Often the delays caused by the communication channels are bigger than those caused by the processing on computational nodes, nevertheless the nodes should not be neglected in the model to achieve a most accurate and realistic system model. A node might have one or more parallel processing channels (single or multi-threaded) with a given processing capacity (instruction rate [Million Instructions per Second(MIPS)]). The number of failed parallel processing channels (threads) can also be defined. Furthermore, a node (including the operating system) can support various scheduling policies (e.g. FIFO or priority scheduling, preemption or no preemption). This is also the layer where the proposed middleware (MW) *Chameleon* resides. Further information in regard to the middleware instances are provided in Sections 4 and 5.

3.5. Application Layer

The applications are composed of concurrent tasks (threads, processes). Thereby, a task can be either time-triggered (periodic, active) or event-triggered (passive, reactive). For each application, several real-time requirements (periods for time-triggered tasks, priorities, deadlines, ...) as well as additional real-time tolerance parameters (maximum allowed missrate of its timing containts, the maximum allowed distance to its timing constraints, ...) can be specified. The mixed-criticality is expressed by the *importance*-Parameter, which defines the importance of a task for the overall application.

The mapping of applications to tasks can be either fine or coarse grain. In a coarse grain mapping, an application is realized by a single task. In an automotive scenario for example, the application *Anti-Lock-Brake* is realized by a task and the application *Navigation* is realized by another task. In a fine-grained mapping, an application is realized by multiple tasks. In an automotive scenario, the application *Anti-Lock-Brake* might be realized e.g. by four tasks (wheel-speed-acquisition, brake-request-acquisition, brake-force-control, brake-light-indicator) while the *Navigation* might be realized by three tasks (GPS-position-detection, route-calculation, traffic-monitoring). Each of these tasks might have different real-time requirements and *importance* values (even within an application, e.g. the brake-light-indicator might be less important/critical

than the brake-force-control). While a coarse grain mapping usually produces less overhead, a fine grain mapping offers more flexibility for adaptation (individual tasks might e.g. be allocated to different nodes).

4. Middleware

A middleware in a CPN is responsible for transparent operation of the distributed network (cf. Section 2.4). It handles all the interactions of applications, sensors and actuators within a single CPS and the entire CPN. It also manages the resources like computation nodes and communication channels to enable distributed operation. Therefore, the middleware is the ideal place to provide self-adaption which affects the entire CPN. Figure 4.1 shows the architecture of *Chameleon*[1] — a middleware for complex mixed-critical CPNs that aims to efficiently handle safety-related non-functional requirements with regard to the mixed-criticality aspect. A comprehensive

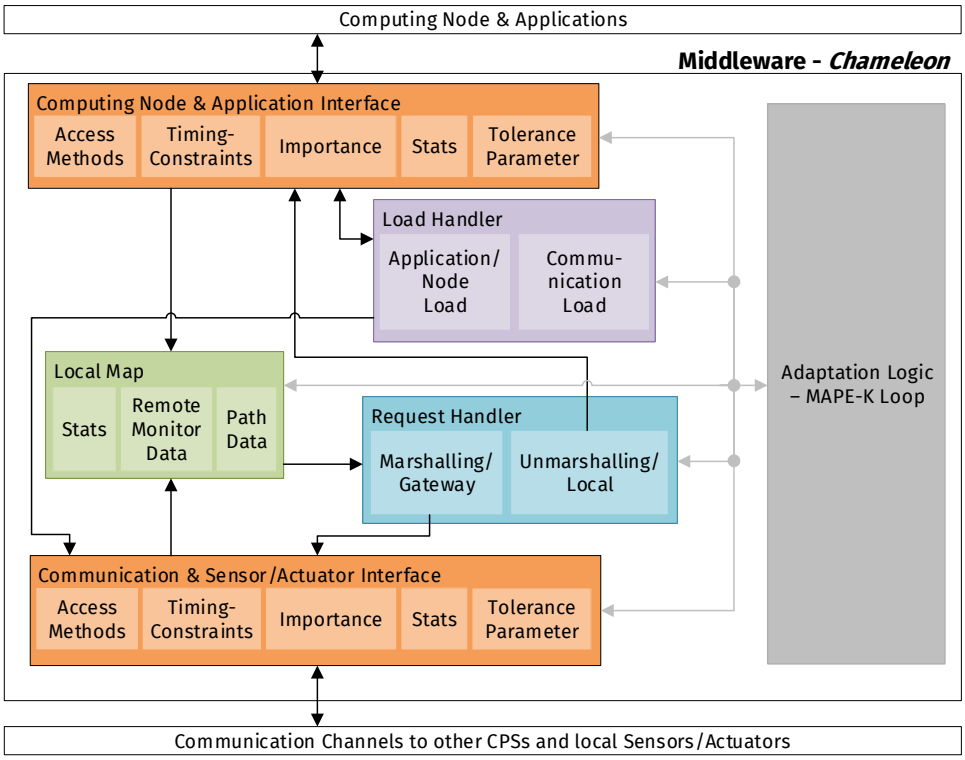


Figure 4.1: Middleware architecture.

approach is used in which all components and interactions within the mixed-critical CPN are subject for adaptation (cf. system model Section 3). Thereby, each *Network*

[1]First principles of the *Chameleon* architecture have been published in [BBK19], while more details can be found in [FB22].

Node in the system has it's own *Chameleon* middleware instance (cf. Figure 3.1). As a result, there there are multiple instances of *Chameleon* within a CPN.

The middleware is structured in a modular way to allow an easy change and modification of its components. It is designed as a framework for the self-adaptation process. The middleware architecture can be divided into two major parts. The basic middleware part shown on the left side of the Figure 4.1 is responsible for the basic middleware operations to provide distributed system interaction and transparency. It consists of several sub-parts: The orange ones represent the interfaces to the other resources of the mixed-critical CPN. They provide access to the defined parameters of applications (given by the system designers, e.g., the definition of timing requirements like deadlines along with the *importance*), nodes (e.g. the node processing capacity, scheduling strategy), communication channels (e.g. the communication capacity, scheduling strategy), sensors and actuators (e.g. the definition of timing requirements along with the *importance*) as defined in the system model (cf. Section 3). The *Load Handler* (purple) handles the load management of applications on the middleware. The *Request Handler* (turquoise) is responsible for the management of messages between the applications and communication side. The *Local Map* (green) takes care of the overlay network routing and distributed monitoring. In the remainder of this Section these components are explained in detail.
The core ingredient of the *Chameleon* middleware architecture is the adaptation logic part shown on the right side of the Figure 4.1 marked in gray. It is responsible for the self-adaptation process. Due to the framework design, different adaptation mechanisms could be integrated. In the frame of this thesis, a MAPE-K loop has been used. It monitors the behavior and adapts the parameter and structure of the system via the other middleware components. Further details on the adaptation are presented in Section 5.

4.1. Computing Node and Application Interface

This module provides access to computing node/operating system data and realizes the interface to the applications. It offers the access methods for application interaction with other applications, sensors and actuators. It deals with the definition of timing constraints for applications and interactions as well as the importance for applications. Also, tolerance parameters can be given and statistic information of the local applications and the local computing node is collected.

General information on applications on the local computing node are given to the middleware by notification events. The major events are:

- ***StartApplicationEvent****(id, importance, period, constraints, toleranceParameters, sensorIds)*:
  This event notifies the middleware *Chameleon* about the start of an application on the local computing node. The application is identified by its *id* and its parameters (properties) are delivered (cf. system model, Table 3.1). The major parameters are the *importance* of the application, required timing constraints

(priority, deadline, reaction time) and *tolerance parameters* (max miss rates, max distances, tuning, max tuning, compression, max compression). For periodic applications, the initial *period* is given. Furthermore, the ids of the periodic and sporadic sensors an application expects data from are delivered. The middleware needs to know to which applications sensor data have to be routed. For reactive sensors, this information can be retrieved from the application sensor data request. For periodic and sporadic sensors however, no such request exists. Therefore this information is delivered here.

- ***UpdateApplicationEvent****(id, importance, period, constraints, toleranceParameters, sensorIds)*:
  This event notifies the middleware *Chameleon* about a parameter change of an application on the local computing node. The parameters are the same as for *StartApplicationEvent*.

- ***StopApplicationEvent****(id)*:
  This event notifies the middleware *Chameleon* that an application has been stopped on the local computing node.

The access methods for applications provide transparency according to the middleware paradigm by hiding communication paths and remote application locations. The following functions are available (cf. Section 6.2.1):

- The following function pairs realize a remote procedure call (RPC):
  DoProcedureCall *targetId callParameters [constraints] /*
  WaitForProcedureCall *[sourceId]callParameters [timeout];*
  DoProcedureReturn *returnParameters [constraints] /*
  WaitForProcedureReturn *[sourceId] returnParameters [timeout]*
  DoProcedureCall calls a procedure in an application running somewhere in the system identified by its target id with the given call parameters. By default, the constraints (priority, deadline, reaction time) of the calling application are used, but can be modified by optional individual constraints of the call.
  WaitForProcedureCall awaits any or a specific procedure call given by the optional parameter soure id and delivers the call parameters. An optional timeout can be defined while waiting. Priority inheritance can be used to assign the constraints from the caller to the called.
  DoProcedureReturn returns a procedure call with the given return parameters. By default, the constraints of the returning application are used, but can be modified by optional individual constraints.
  WaitForProcedureReturn awaits any or a specific procedure return given by the optional parameter source id and delivers the return parameters. An optional timeout can be defined while waiting.

- The following function pairs realize a remote method invocation (RMI). It works analog to the RPC, but for methods of objects instead of procedures:

DoMethodInvocation *targetId callParameters [constraints] /*
WaitForMethodInvocation *[sourceId] callParameters [timeout];*
DoMethodReturn *returnParameters [constraints] /*
WaitForMethodReturn *[sourceId] returnParameters [timeout]*

- The following function pairs realize a message passing. Instead of calling a procedure or invoking a method, a message is passed to the target and it is possible to submit an answer by *DoMessageRetur*n. Otherwise, it works analog to the RPC and the RMI:
  DoMessagePassing *targetId message [constraints] /*
  WaitForMessage *[sourceId] message [timeout];*
  DoMessageReturn *message [constraints] /*
  WaitForMessageReturn *[sourceId] message [timeout]*

- The following function pair requests data from a reactive sensor somewhere in the system identified by its sensor id and waits for the data arrival:
  DoRequestSensorData *sensorId [constraints] /*
  WaitForSensorData *sensorId data [timeout]*
  In case of periodic or sporadic sensors the *WaitForSensorData* function is used without *DoRequestSensorData*. Constraints and timeout are the same as for the functions above.

- The following function sends data to an actuator somewhere in the system given by its actuator id. Constraints are the same as for the functions above:
  DoDataToActuator *actuatorId data [constraints]*

- This following function pair sends a data stream in portions with the given stream period. Constraints and timeout are the same as for the functions above:
  DoDataStream *targetId streamPeriod streamPortions [constraints] /*
  WaitForDataStreamPortion *[sourceId] streamPortions [timeout]*

The interface also collects statistic information of the local node and its applications like the nominal computational capacity of the node, current node load and scheduling scheme, the current computational and data demand of the applications as well as the current miss rates and distances of the application timing constraints. This information is an important basis for the adaptation process, see Section 5. Furthermore, node and application parameters (node scheduling scheme, application constraints, periods, tuning factors and data compression) can be modified by the interface.

4.2. Communication and Sensor/Actuator Interface

This interface provides access to the communication channels and sensors/actuators attached to the local computing node. For the communication channels, it offers the basic access methods to send and receive data via a specific channel given by its channel id. Herewith, unicast and broadcast transmissions are supported. It also

collects statistical information about the communication channels like the nominal communication capacity of the channels, the current channel load and failure rates. For the local sensors and actuators, the basic access methods to read and write sensor/actuator data are provided. Furthermore, importance, constraints (priority, deadline, reaction time) and tolerance parameters (max miss rates, max distances) can be specified. For periodic sensors the period can be given.

4.3. Local Map

The *Local Map* takes care of the overlay network routing and distributed monitoring. The low-level routing of the communication channels (how to send a message from one member of the channel to another one) is done by the channels itself. However, the middleware instances also have to cope with routing. On the one hand, a middleware instance needs to know where its target applications are currently located (on which network node the applications reside) and how to reach them (by which local communication channel data can be transmitted to the network node). This is also true for the sensors and actuators. On the other hand, a middleware instance might also have to act as a gateway. In an example configuration as shown in Figure 4.2, *Chameleon* middleware instance A (MW_A) can only reach *Chameleon* middleware instance C (MW_C) when *Chameleon* middleware instance B (MW_B) operates as gateway (think of communication channel $Comm_1$ being e.g. the internet and communication channel $Comm_2$ being a local CAN bus). Therefore, the middleware instances establish an

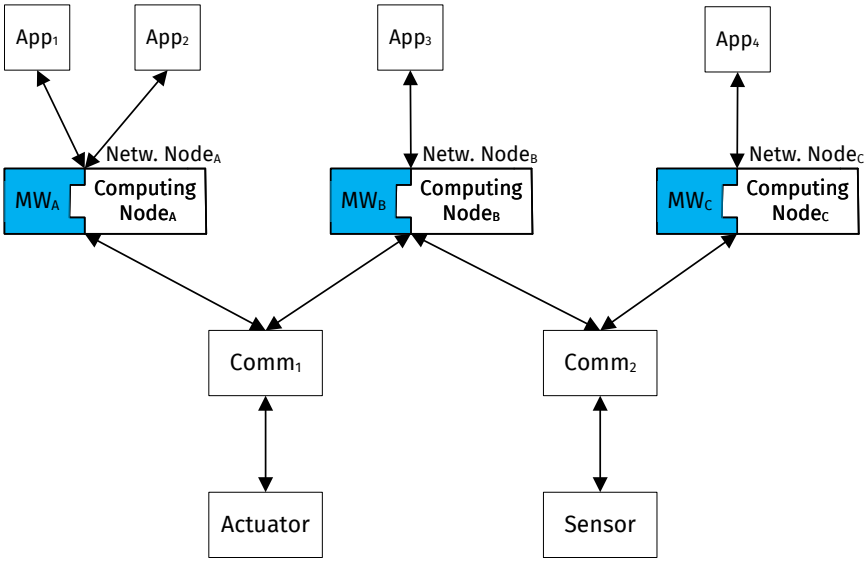


Figure 4.2: Example network configuration with three middleware instances.

overlay network. The *Local Map* in each middleware instance takes care of the routing information necessary on middleware level.

A local decentralized approach has been chosen for the realization of the overlay network routing. This is the case since a global centralized approach could represent a single-point-of-failure and therefore harm the adaptation process in the presence of failures, thus causing a severe threat. The *Local Map* learns the locations and paths to applications, other network nodes, sensors and actuators by evaluating incoming messages and storing the retrieved information in a map for later use. This is done by looking at the messages and requests received from remote locations (cf. Section 4.5). Analyzing the communication path of an incoming message delivers the route to the source (application, middleware). Additionally, "hello" messages for local components (applications, sensors, actuators) are sent at the start (e.g. the *StartApplicationEvent*, see Section 4.1) and on request of other middleware instances (if e.g. a path to a target is currently not known by a middleware instance). By that the local map is filled with information on how to reach a target. Furthermore, communication statistics (e.g. average relative communication times, useful to find best routes to a target) and distributed monitoring data (e.g. status and parameter data of remote applications, nodes and communication channels (cf. monitoring section 5.1) are stored as well in the *Local Map*. Finally, the mapping of periodic and sporadic sensors (which are not triggered by an application, but send data on their own) to applications is stored there. Figure 4.3 sketches the structure and information flow of the *Local Map*. Internally, it can be divided into the *Target Map*, where the target information is stored and the *Sensor Map*, where the sensor/application mapping is stored. Tables 4.1 and 4.2 provide an overview of their contents. The update component for target/sensor learning and the query component for information retrieval completes the *Local Map*.

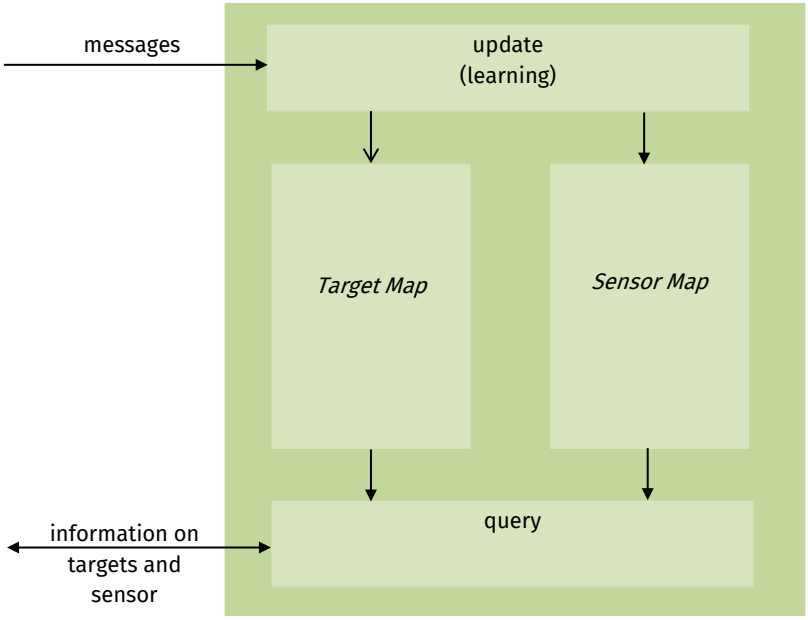


Figure 4.3: Structure and information flow of the *Local Map*.

targetId	id of target, map index
type	type of the target (application, middleware, sensor, actuator)
mwId	id of the middleware where the target resides
path	communication path to the target
commStats	communication statistics (e.g. communication time) to the target
monData	remote monitoring data (current status and parameters, e.g. period, load, ...)
timestamp	last update time (e.g. used for life signs)

Table 4.1: Structure of a *Target Map* entry.

appId	id of the application, map index
sensorId	id of the sensor the application expects data from

Table 4.2: Structure of a *Sensor Map* entry.

4.4. Load Handler

The *Load Handler* handles the load management of applications and sensors on the middleware. It uses the *Computing Node and Application Interface* as well as the *Communication and Sensor/Actuator Interface* to access and modify the computational and communication load caused by applications and sensors on the local node and on the communication channels of the middleware. Thereby, it offers the following adaptation actions:

1. **Start**: Starts an application on the local node.
2. **Stop**: Stops an application on the local node.
3. **Relocate**: Relocates an application from somewhere in the system to the local node.
4. **Remove**: Removes a local application from the system.
5. **Tune**: Tunes an application/sensor of the local node by modifying its constraints and parameters (e.g. priorities, periods, deadlines).
6. Set data **compression**: Compress the messages/sensor data of an application/sensor on the local node.
7. Modification of **scheduling strategy**: Changes the scheduling parameters and or schemes of the local node or local communication channel(s).

Using these actions, the adaptation logic of the middleware instance can trigger the local part of a global adaption goal (c.f. Sections 5.3 and 5.4).

4.5. Request Handler

The *Request Handler* is responsible for the management of messages between the application and communication side. Principally, requests are the access functions offered by the *Computing Node and Application Interface*, see Section 4.1. These can be RPC, RMI, message passing or sensor/actuator data transfer. The *Request Handler* determines the origin and the destination of such a request with the help of the *Local Map*. In general, four scenarios are possible which are illustrated in Figure 4.4 (derived from the previous Figure 4.2):

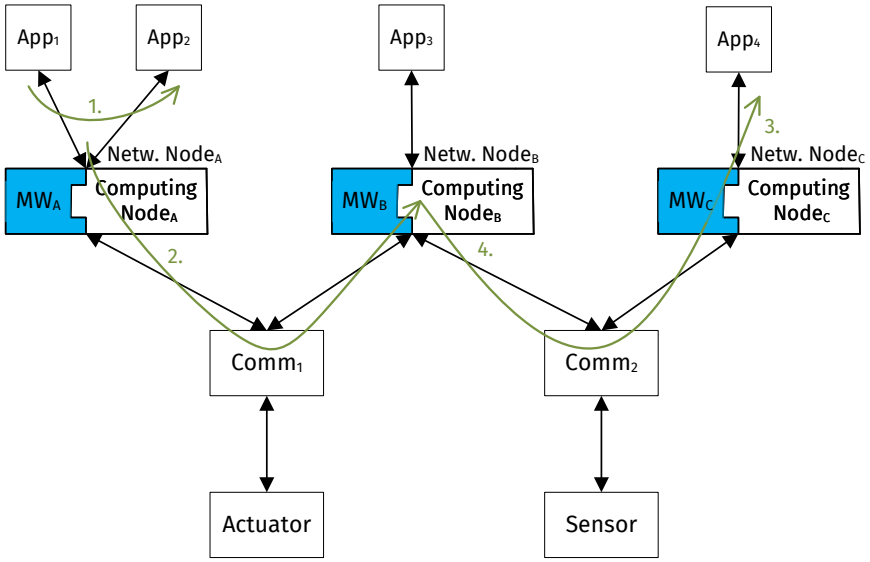


Figure 4.4: Request handler communication scenarios.

1. **Source and destination are both local:**
   In that case, no data has to be send over the network. The request can directly be forwarded from the source to the destination.

2. **Source is local and destination is remote:**
   Here, the local request has to be send over the network to its remote destination. To be able to do so, the request (e.g. a RPC) has to be transformed into a send-able message. This process is called *Marshalling*. Optional data compression (cf. Section 4.4) is also applied here.

3. **Source is remote and destination is local:**
   In that case, a message received from the network has be transformed back to the initial request. This process, which is the opposite of *Marshalling*, is called *Unmarshalling*. Optional data uncompression also applies here. Then, the request is delivered to its local destination.

4. **Source and origin are both remote:**
   Here, the local middleware instance simply acts as a gateway (cf. Section 4.3). The received message/request has only to be forwarded towards the destination.

The messages received in cases 3 and 4 are also used by the *Local Map* to learn the route to the source of the message (cf. Section 4.3).

5. Adaptation

Autonomous adaptation is a key feature for MC-CPN to cope with the inherent complexity of such systems due to the mixed-criticality itself and the ever increasing number of cooperating and interacting systems within uncertain operating conditions. As already mentioned in the previous Section 4, the common MAPE-K feedback loop architecture (cf. Section 2.3.3) is used to realize the desired self-adaptation capabilities in the MC-CPN. The MAPE-K loop, which was first introduced by IBM in the frame of *Autonomic Computing*, can be seen as an architectural blueprint for self-adaptation mechanisms (cf. Section 2.3.3). Therefore, it can be realized in many different ways . In general, it consists of four components for (i) **M**onitoring system and environment, (ii) **A**nalyzing monitoring data for required adaptations, (iii) **P**lanning adaptations, and (iv) **E**xecuting them. In addition, the four components share a **K**nowledge base. A possible representation of the analyzing, planning component and the knowledge base is a LCS (cf. Section 2.3.4). LCS-based approaches are popular in *Organic Computing* (cf. [STH16; Ste+20]) and have been applied for some approaches in the CPS domain [Ste+17a; STH16; TH11; HPH20].

In comparison to other online learning approaches, LCS offer a practical and feasible computational complexity. Therefore, training can be performed on devices with low computational resources and decisions can be made in real-time, which are both essential requirements for the CPN adaptation process. Thus, the LCS mechanism has been selected for the MAPE-K loop in this approach. LCS uses a set of *rules* that represent potential adaptations schemes. A rule thereby mainly consists of a condition clause, an action clause and an expected reward. Based on the measured *reward*, i.e., the effectiveness of an action, that was observed after applying a certain rule, LCS learn and select suitable rules for future adaptations. Figure 5.1 shows a typical LCS-based MAPE-K feedback loop. The monitor receives context information about the state of the mixed-critical CPN, pre-processes it, and forwards relevant information to the analyzer. The analyzer contains the *rule engine*, which is responsible for rule evaluation and reward calculation. It compares the current system condition to the condition clauses of the *rule set* stored in the knowledge component. All rules that are currently applicable, i.e., all rules where the condition clauses evaluate to true, are included in the *match set*. The planner collects all action clauses from the match set into the action set. Then, it selects the action that is expected to lead to the highest reward[1]. In the next step, the execution component controls the deployment of the rules. Finally, the analyzer determines the measured reward and updates the rule set accordingly. An LCS-based feedback loop may additionally contain a rule

[1]This can be done value based like in classical LCS or precision based (where the measured reward best matches the expected reward) in extended learning classifier systems (XCS [ST21; Wil95; WS21])

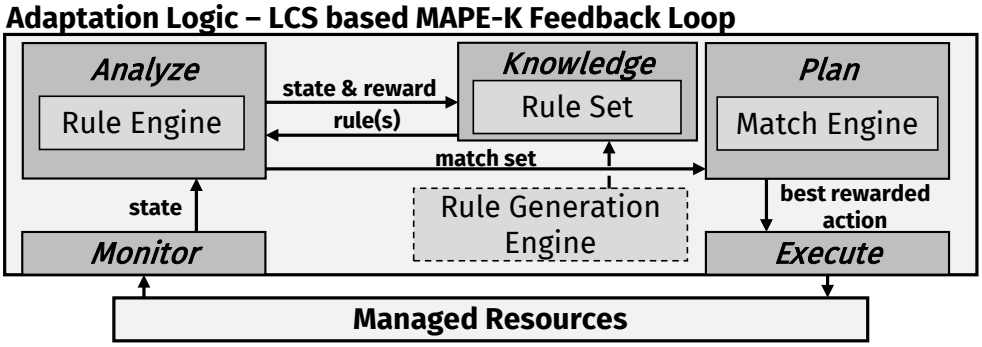


Figure 5.1: Typical architecture of an LCS-based MAPE-K feedback loop. The analyzer's rule engine adds rules that are applicable in the current context to the match set. The planner's match engine selects the action of the most promising rule for execution. A rule generation engine may evolve rules at runtime.

generation engine, which evolves the rule set automatically at runtime, e.g., with genetic algorithms [SW09; UM09]. Due to the distributed nature of a CPN, the LCS approach has to be extended to a distributed LCS (DLCS) to operate in the environment (cf. system model described in Section 3.1: Each network node has its own middleware instance and thus LCS). This results in different scopes for conditions and actions (cf. Sections 5.2.2 and 5.3). Furthermore, interferences between actions executed on different nodes in the distributed environment have to be taken into account (cf. Section 5.2.3).

In the following, the stages used in the distributed MAPE-K feedback loop are explained in detail.

5.1. Monitor

The *Monitor* samples data from the local node, each local communication channel (separately for direction in or out) and each local application and derives further information from the data as can be seen in Figure 5.2. Thereby, the monitored data includes the values of the components properties (cf. Section 3). Table 5.1 gives an overview of the most important items sampled by local monitoring from the system components. For instance, the capacity and remaining capacity of the network nodes and communication channels are observed, along with the processing and communication demands of applications.

Source	Item	Remark
Node	capacity	The nominal processing capacity of the node in instructions/s
	load	The load of the node in %

	remaining capacity	The remaining processing capacity of the node in instructions/s (derived from load an capacity)
	scheduling policy	The current and available scheduling policies of the node
Comm. Channel	direction	Input or output channel
	duplex	The transmission mode (simplex, half-duplex, duplex)
	capacity	The nominal data transmission capacity of the channel in bits/s
	load	The load of the channel in %
	remaining capacity	The remaing communication capacity of the channel in bits/s (derived from load and data rate)
	scheduling policy	the current and available scheduling policies of the channel
Application	*Parameters:*	
	importance	The importance of the application
	priority	The priority of the application
	period	The period of the application in s
	deadline	The deadline of the appication in s
	reaction time	The reaction time of the application in s
	tuning factors	The current tuning factors for priority, deadline, reaction time and period
	max tuning	The maximum tuning factor
	compression	The current data compression factor
	max compression	The maximum compression factor
	max miss rates	The maximum allowed miss rates for deadline, reaction time and period
	max distance	The maximum allowed distances to deadline, reaction time and period
	Current demands:	
	node demand	The processing demand of the application in instructions/s
	comm. channel demand	The communication demand of the application in bits/s (separately for in and out direction)
	Current real-time performance:	
	miss rates	The current miss rates for deadline, reaction time and period
	distances	The current distance to the deadline, reaction and period in s

Table 5.1: Major items for a component sampled and derived by local monitoring.

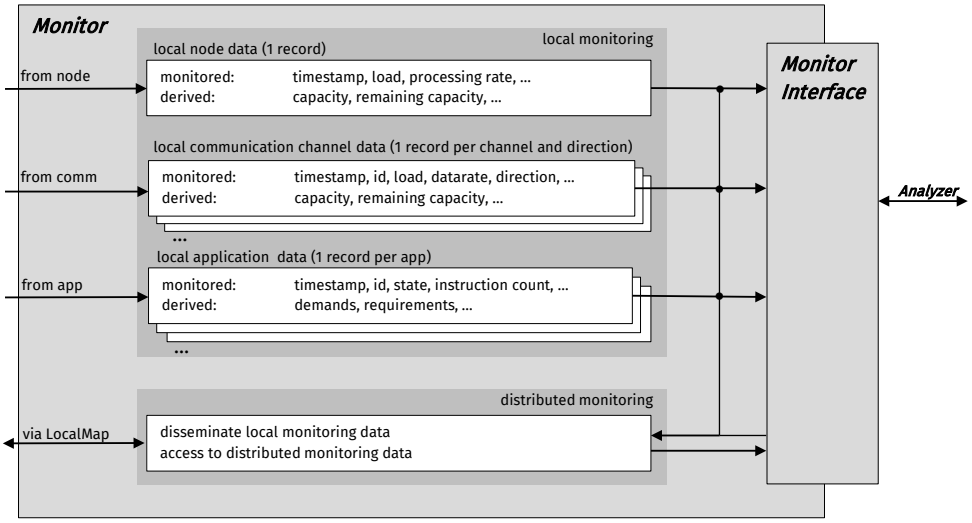


Figure 5.2: Monitor architecture.

Thereby, the monitoring period(the period in which parameters of the components are monitored) can be individually defined based on the properties of the component (e.g. period of the application, performance of nodes and communication channels, ...) and can be modified by the *Analyzer* via the *Monitor Interface*. Furthermore, this interface informs the *Analyzer* of new monitored data.

In addition, due to the end-to-end nature of the requirements, a global view of the system is required to decide if adaptation is necessary. This is realized by distributed monitoring (cf. Figure 5.2). It is performed by disseminating important local parameters and status data periodically (with a multiple of the monitoring period) or on event (e.g. the change of an application parameter) via the network. The more frequently the data is distributed in the system, the more up-to-date the remote monitoring data is but the higher the communication overhead for the dissemination will become. The monitoring data from remote locations are accessed via the *Local Map* (cf. Section 4.3) from the local *Monitor*.

Thereby, various remote parameters and status data like timing requirements including the importance of remote applications, processing and communication capacity of remote nodes, processing and communication demand of remote applications, health values (cf. Section 5.2.1) of applications, nodes and communication channels etc. can be retrieved via the *Local Map* component in addition to the local monitoring data. Since the communication and computational capacities of remote nodes are of particular interest for the adaptation, the communication parameters and status are considered in the context of the nodes status data. Therefore, the parameter and status data of all communication channels of a node are subsumed and condensed. This reduces the communication overhead for the exchange of remote monitoring data. Table 5.2 shows the most important items available from remote monitoring. Thereby, communication

channel information like the remaining capacities are included in the remote monitoring data of the corresponding network node. The update period specifies the frequency at which remote monitoring data from the components is distributed within the system.

Source	**Item**	**Remark**
Node	update period*	The update period of the remote node data
	node capacity	The nominal processing capacity of the node in instructions/s
	best comm. channel capacity	The best nominal data rate for the communication channels of the node in bits/s (separately for in and out direction)
	worst comm. channel capacity	The worst nominal data rate for the communication channels of the node in bits/s (separately for in and out direction)
	remaining processing capacity	The remaining processing capacity of the node in instructions/s (derived from load an capacity)
	best remaining communication capacity	The best remaining communication capacity for the communication channels of the node in bits/s (separately for in and out direction)
	worst remaining communication capacity	The worst remaining communication capacity for the communication channels of the node in bits/s (separately for in and out direction)
Application	update period*	The update period of the remote application data
	Parameters:	
	importance	The importance of the application
	priority	The priority of the application
	period	The period of the application in s
	deadline	The deadline of the appication in s
	reaction time	The reaction time of the application in s
	tuning factors	The current tuning factors for priority, deadline, reaction time and period
	max tuning	The maximum tuning factor
	compression	The current data compression factor
	max compression	The maximum data compression factor
	max miss rates	The maximum allowed miss rates for deadline, reaction time and period
	max distance	The maximum allowed distances to deadline, reaction time and period
	Current demands:	
	node demand	The processing demand of the application in instructions/s
	comm. channel demand	The communication demand of the application in bits/s (separately for in and out direction)

	Current real-time performance:	
	miss rates	The current miss rates for deadline, reaction time and period
	distances	The current distance to the deadline, reaction and period in s

Table 5.2: Major items for a component available from remote monitoring (* the update period can be used to determine the limit for life sign age)

This periodic dissemination of remote monitoring data is also be used as a life sign for nodes and applications. Thus, it allows to detect crashes. The regular update period (cf. Table 5.2) can be taken to calculate the life sign age limit beyond which a node or application is considered to be crashed.

5.2. Analyze, Plan and Knowledge

The Analyzer can access monitored data via the *Monitoring Interface* as depicted in Figure 5.2. The retrieved highly specific monitoring data is transformed into abstract *health values* by preprocessing (cf. Section 5.2.1). These deducted *health values* are disseminated via the *Monitoring Interface* and the network as additional distributed monitoring data. Afterwards, the system condition can be analyzed to trigger adaptation if necessary. In this approach, the *Analyze* and *Plan* component as well as the *Knowledge* base are realized with a LCS. Thus, an important challenge is the formulation of a rule set. Therefore, the rule language *Rango* – a generic and flexible rule language (cf. Section 5.2.2) – has been developed. Furthermore, the distributed nature of an CPN with its multiple middleware instances (cf. system model in Section 3.1) leads to interferences between actions executed on different computing nodes in the distributed environment. Those interferences needs to be taken into account e.g. by the rule evaluation (cf. Section 5.2.2), reward calculation (cf. Section 5.2.3) and execution (cf. Section 5.3).

5.2.1. Health Values

In order to be able to specify generalized and uniform conditions for the LCS rules of the adaptation mechanism, an additional layer of abstraction is introduced. Abstract and normalized *health values* $(-\infty, 1]$ provide a unified concept for the definition of the components and system condition. This abstraction additionally allows the developed middleware framework and adaptation mechanism to be easily configurable to the specific requirements of diverse application scenarios, since the the *health values* can be derived from any combination of data retrieved from system monitoring by an analyze preprocessing stage. Statistics (e.g miss rates and average distances to constraints) of applications and sensors can be used as well as the status data of the applications (e.g. processing and communication demand), nodes (e.g. remaining

processing capacity) and communication channels (e.g. remaining capacity for sending and receiving messages)[1].

Thus, the system condition is expressed by multiple of such *health values*. Based on the observation of these *health values*, adequate adaptation measures can be deducted and expressed by rules. This is possible since the *health values* are derived and normalized in such a way that they can be used as margins to define the so-called *target* and *acceptance space* [Sch+10]. The conditions for the rules can then be formulated such that adaptation is triggered when the system's state exits the *acceptance* or *target space*. The *target space* specifies the behavior range in which the designer wants the system to operate. The *acceptance space* gives the still tolerable behavior range. Thereby, a *health value* from $[0, 1]$ marks the *acceptance space* (acc) with 1 as the best possible and 0 as the least acceptable *health value*. The *target space* (tar) is a subset of the acceptance space and is defined by *health values* greater than zero ($[\alpha, 1]$ with $\alpha > 0$). This is illustrated in Figure 5.3. Values below zero mark unacceptable behavior.

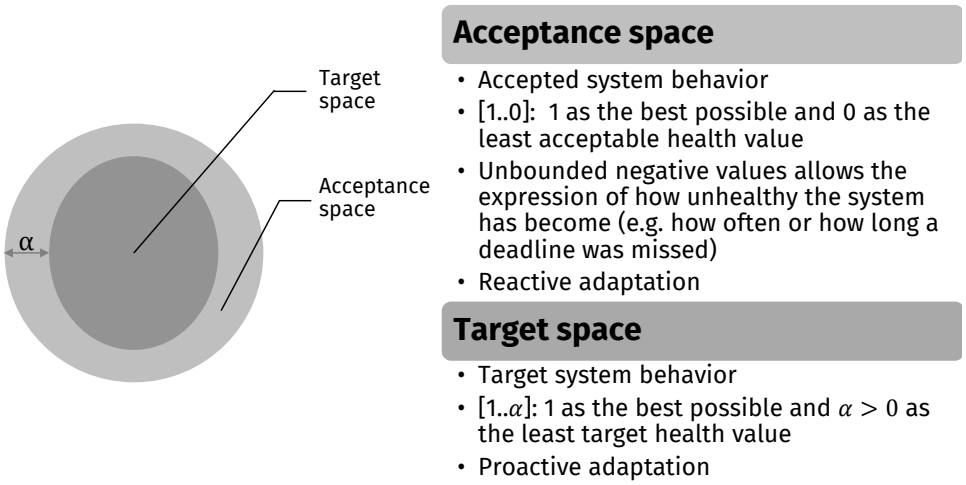


Figure 5.3: Health values can be used as margins to define the *target* and *acceptance* space.

$$health_{range} = \underbrace{-\infty <}_{unhealty} \overbrace{\underbrace{0 \ldots}_{healthy} \overbrace{\alpha \ldots 1}^{tar}}^{acc} \tag{5.1}$$

Having unbounded negative values allows the expression of how unhealthy the system has become (e.g. how often or how long a deadline was missed). As already

[1]Other constraints like energy were not in the focus of this thesis, but could also be mapped to *health values*, see future work 9.2.

mentioned above, adaptation can be triggered if necessary to stay in the bounds of the *acceptance* or *target space*. This makes the system robust. Furthermore, utilizing these two spaces to define desired behavior enables the realization of both reactive and proactive adaptation, depending on which bound violation triggers the adaptation. Proactive adaptation involves making adjustments to the system before problems arise, specifically when the bounds of the *target space* are violated. In contrast, reactive adaptation responds to issues that have already occurred (e.g a missed deadline), and are therefore triggered by violations of the *acceptance space* bounds.

Moreover, the system adaptation goals can be altered by adapting the limits or the derivation of the *health values*. This makes the system flexible by shifting the *target* and *acceptance space*.

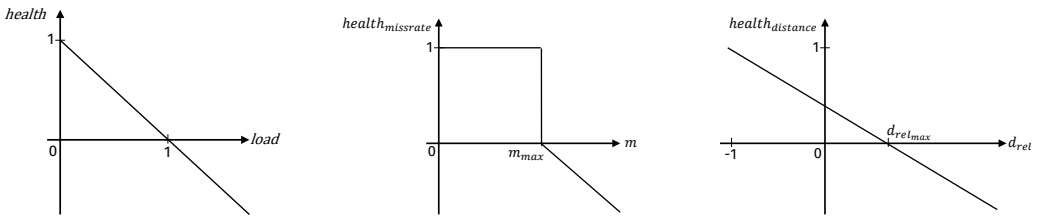
In the following, the derivation of *health values* in the normalized range for applications, nodes, communication channels and the middleware based on load and timing (real-time) constraints are presented. Furthermore, Section 5.4, which serves as a summary of this chapter, presents an example (cf. Table 5.6) illustrating how *health values* can be utilized to infer the conditions of components and the overall system, and subsequently deduce potential adaptation measures.

Node and Communication Channel Health The health of nodes and communication channels can be derived using the load of the node or communication channel. Thus, the node and communication channel health can be calculated as a load health.

Let's assume we have the load of the node $load_{node}$ and the load of the communication channel $load_{comm}$ provided from system monitoring. Thereby, the load ranges from 0 (unloaded) over 1 (fully loaded) to > 1 (overloaded). The corresponding *health values* for the node and communication channel can then be defined as (cf. Figure 5.4a):

$$health_{node} = 1 - load_{node} \tag{5.2}$$

$$health_{comm} = 1 - load_{comm} \tag{5.3}$$



(a) Load health **(b)** Missrate health **(c)** Distance health

Figure 5.4: Derivation of *health values* for applications (missrate & distance health), nodes and communication channel (load health).

Application Health The *health value* of applications can be derived using timing constraints. System monitoring provides percentage timing missrates m ([0,1]) and average distances d to timing constraints. Also, each application defines allowed maximum missrates m_{max} (e.g. hard real-time, soft real-time) and allowed maximum distance $d_{rel_{max}}$ (relative maximum distance to the constraint, ≥ 0) to the constraints as real-time tolerance parameters (cf. Table 3.1) thus defining margins for the *acceptance space*.

In a concrete scenario of a vehicle, the brake controller might have a deadline of 30 ms and is not allowed to miss this deadline more often than in 5% of its execution cycles ($m_{max} = 0.05$) and not by more than 3 ms ($d_{rel_{max}} = 0.1$).

Such missrates and average distance can easily be transformed into two separate *health values*. The missrate based *health value* is 1 as long as the missrate is below or equal to the allowed missrate. As soon as the missrate gets above the allowed missrate, the *health value* gets negative (cf. Figure 5.4b):

$$health_{missrate} = \begin{cases} 1 & \text{if } m \leq m_{max} \\ m_{max} - m & \text{else} \end{cases} \tag{5.4}$$

The distance to constraint calculates by *actual value* − *constraint*. Thus, distance to constraint ranges from $-constraint$ (best case, e.g. if no time was needed at all for a timing constraint) via 0 (exactly hit the constraint) to > 0 (missed the constraint by that value). To normalize this, the relative distance to constraint is used:

$$d_{rel} = \frac{d}{constraint} \tag{5.5}$$

The distance based *health value* then calculates to (cf. Figure 5.4c)):

$$health_{distance} = \frac{d_{rel_{max}} - d_{rel}}{d_{rel_{max}} + 1} \tag{5.6}$$

The overall timing constraint health for an application is then given by

$$health_{app} = \min(health_{missrate}, health_{distance}) \tag{5.7}$$

In the above mentioned vehicle scenario, $health_{missrate}$ calculates to 1, as long as the actual missrate is not above 5%. Otherwise it becomes negative. Whilst the deadline itself is not exceeded by more than 3 ms, $health_{distance}$ stays between 1 and 0. Otherwise it also gets negative.

Furthermore, an application can have up to three timing constraints (period, deadline, reaction time (cf. Table 3.1)). If more than one of them is defined, the worst value is used.

Middleware Health The middleware health $health_{mw}$ can be calculated as a derivate of the health of the middleware's local applications $health_{apps_{local}}$[1].

$$health_{mw} = \min(health_{apps_{local}}) \tag{5.8}$$

5.2.2. Rule Language – *Rango*

While the fundamental architecture of LCS is well-established, an important challenge remains: The rule set must be formulated. Therefore, the rule language *Rango* – a generic and flexible rule language – has been developed in the frame of this thesis[2]. Usually, LCS use bit strings to codify rules [SW09; UM09]. For instance, "11→1" could be a representation of "`IF` carFacesObstacle `AND` carInMotion `THEN` brake". Analogously, another rule would be "01→0", which would be equal to "IF !carFacesObstacle AND carInMotion THEN !brake".

While bit strings might be feasible to use as representation of rules for this approach, it is not particularly beneficial, since the complexity of typical MC-CPN use cases would lead to lengthy bit strings that require a significant coding effort and are cumbersome to read and write. Instead, a flexible language called *Rango* is being developed to fulfill various goals and accommodate the dynamic nature of the rules, allowing for greater adaptability and ease of rule set formulation in an intuitive way. It offers maximum flexibility to define abstract and generic rules independent of a specific use case as well as application-specific rules, if these are desired.
As depicted in Figure 5.5, rules written in *Rango* are stored in a `.rul` file. This file is automatically parsed into an internal binary representation that can be used as a rule set of the feedback loop without any modifications. Thereby, *Rango* has three major advantages for specifying adaptation behavior in MC-CPNs. First, it includes a large set of pre-defined CPN/CPS keywords (cf. Table 5.3). These specific keywords are re-usable for many CPN/CPS use cases. Second, *Rango* is well-integrated with the learning component. The rules specified in *Rango* are automatically transferred into a binary representation that is usable by a LCS without any modifications as base of the learning process. Third, this approach paves the way towards explainable artificial intelligence (XAI) [Hag18; MZR21] by improving traceability and explainability of learning as it offers the option to export a rule set modified by learning to a `.rulx` file in *Rango*'s human-readable form.

In the following, *Rango*'s features are described by applying it to a simplified mixed-critical CPN automotive scenario. After presenting the scenario, *Rango*'s syntax, its CPS-specific features, its rule files, and the evaluation of theses rules are explained.

[1] Of course also the health of the local node and communication channels could be included in the middleware health. However, during evaluation runs this simple approach has shown to be suffient in the frame of the thesis.

[2] The basic ideas of Rango have been first published in [Fei+22].

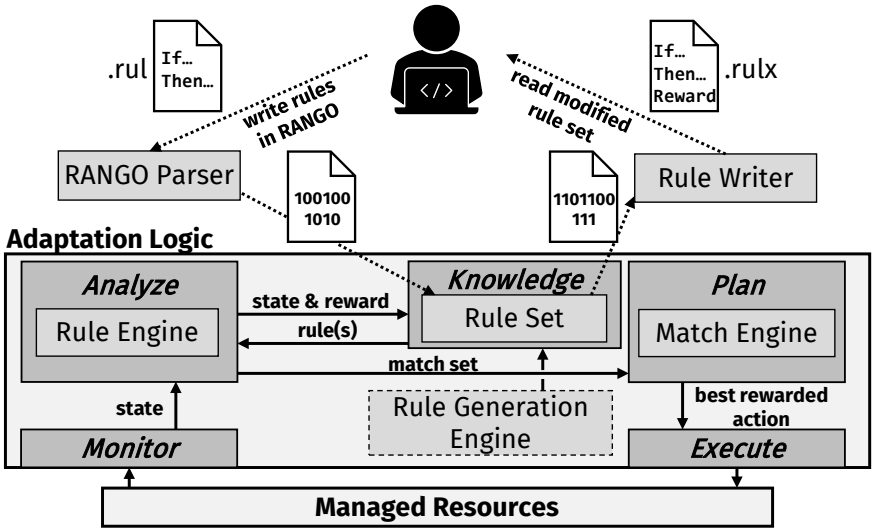


Figure 5.5: *Rango* is a language that allows to formulate the rule set for a LCS in a human-readable format. These rules are parsed and automatically integrated into the learning component. *Rango* also offers a rule writer, which exports rules from the learning component into a human-readable format.

Automotive Scenario The mixed-critical CPN scenario from the automotive domain is shown in Figure 5.6. It is a simplified configuration of the system model presented in Figure 3.1 of Section 3. For easier understanding, the sensors and actuators of the system are neglected. In the scenario, a vehicle has several electronic control units (ECUs), i.e., network nodes with a certain computational power. Each ECU can be responsible for various car functions such as brake control, cruise control, or steering. Apart from those safety-related functions, the ECUs also control comfort and assistance functions including navigation or infotainment. We refer to these functions as *applications* in the following. Each application has predefined timing constraints which should be complied to at all time. The communication among ECUs is realized via a typical vehicular *communication channel* (e.g., CAN, time-triggered ethernet, or FlexRay).

During the journey, the number of active and required applications in the car varies. For instance, the driver may start the navigation system due to a road closure. This potentially leads to an ECU overload if the ECU on which the navigation application is started does not have sufficient computational power to cope with the increased demand. As a consequence, the applications on this ECU may fail to meet their requirements. Thus, the local ECU and the corresponding applications become unhealthy. The overall goal of the adaptation in this scenario is to maintain a good *health value* within the target or acceptance space for each component. If the communication channel within the system is e.g. still healthy, a relocation of the unhealthy application to a healthy ECU might be a suitable adaptation action to achieve an overall healthy system. In

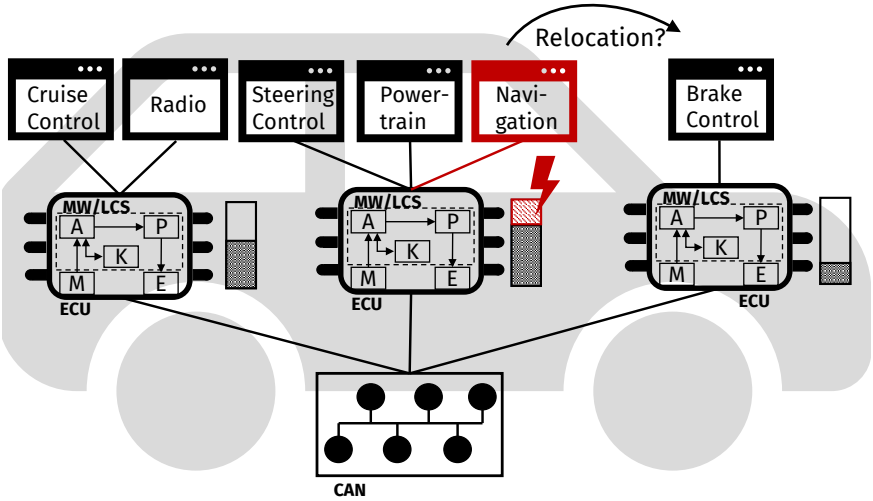


Figure 5.6: Mixed-critical CPN example from the automotive domain. A car contains several ECUs that execute applications with various level of safety-criticallity. After the driver activates the navigation system, the corresponding ECU has to migrate an application to another ECU due to a lack of processing power. It is shown how *Rango* can be used to formulate this desired adaptation behavior.

the following, it is shown how this adaptation behavior can be easily specified with *Rango*.

Syntax In this section, the features of *Rango* are described based on the relocation of an unhealthy application adaptation action in case of an overload in the system as explained in the scenario of the previous section. The complete *Rango* context-free grammar describing the syntax of the language in Extended Backus-Naur Form (EBNF) is available in the Appendix A. The adaptation behavior that is desirable in the example scenario — a relocation of an unhealthy application — can be specified with the following rule in *Rango*:

```
If RelocationMightBeUseful
Then RelocateLocalUnhealthyApp
```

In general, a rule consists of a *condition* and an *action*. Conditions and actions can either be specified directly or defined and referenced by name as shown in the *Rango* code snippet above. Here, the name of the condition is `RelocationMightBeUseful` and the action's name is `RelocateLocalUnhealthyApp`. The reference by name allows to use an action or condition multiple times in a rule set without having to rewrite it. In addition, this increases the efficiency of rule evaluation within the feedback loop, since a condition/action that is used multiple times only needs to be evaluated once.

A condition or action needs to be defined before referencing it by name. In this example, the `RelocationMightBeUseful` condition is defined first. This condition

checks whether unhealthy applications are running on the local ECU while (i) the communication is still healthy and (ii) there are other suitable ECUs in the car to which the unhealthy applications can be migrated. In *Rango*, this condition is formulated as follows:

```
DefineCondition RelocationMightBeUseful :
Cardinal (LocalUnhealthyApps) > 0 And
Cardinal (LocalUnhealthyNode) > 0 And
Cardinal (NonLocalSuitableNode) > 0 And
Cardinal (LocalUnhealthyComm) = 0
```

This example shows that a condition consists of conjunctive links (`And`) of *sub-conditions*. Sub-conditions always operate on *sets*. A set can contain applications, nodes, or communication channels. The `Cardinal` keyword delivers the cardinality of a set. In addition to their cardinality, the maximum, minimum, or average value of one of the set's attributes can be used for comparison. In the code snippet above, the cardinality of four sets (`LocalUnhealthyApps`, `LocalUnhealthyNode`, `NonLocalSuitableNode`, `LocalUnhealthyComm`) is used to specify sub-conditions. The overall condition `RelocationMightBeUseful` therefore evaluates to true if the first three sets are non-empty while the last one is empty.

Rango offers two options for the definition of sets. First, sets can be defined by directly referencing applications, nodes, or communication channels by name. Second, sets can be defined via *queries*, e.g., by selecting all applications with a *health value* smaller than 0. Defining sets via queries makes it possible to keep the rules independent of specific applications. Instead, *Rango* constructs these sets dynamically based on the respective query. In a query, the attributes of a node, application, or communication channel can be compared to arbitrary values or their maximum/minimum value can be requested. Two exemplary set definitions with queries are:

```
DefineSet LocalUnhealthyApps :
App Local Where Health < 0

DefineSet NonLocalSuitableNode :
Node NonLocal Where
Capacity >= Demand (LocalUnhealthyApps),
Health Max
```

Here, the first set (`LocalUnhealthyApps`) includes all local unhealthy applications, i.e., those with *health values* below 0. The second set (`NonLocalSuitableNode`) contains the most healthy non-local nodes with sufficient remaining computing power[1]. Similar to conditions and actions, a set can be used several times by referencing to its name. A set that is used multiple times is only constructed once per rule evaluation period.

[1] If there is more than one element in the `LocalUnhealthyApps` set, the demand of the first application of this set is used for comparison. It is also possible to use the demand of all applications in the set with the `All` keyword.

So far, all of *Rango*'s features that are required to specify the condition are introduced. Now, let's have a look at the definition of the action (`RelocateLocalUnhealtyApp`):

```
DefineAction RelocateLocalUnhealthyApp :
Relocate LocalUnhealthyApps
To NonLocalSuitableNode
```

It can be seen that this action definition re-uses the sets already defined above. An action can be applied either to all elements of the set or to one element. This is realized by either using or omitting the `All` keyword. Without the `All` keyword, only the first element of the `LocalUnhealthyApps` set is moved to the first element of the `NonLocalSuitableNode` set. If the `All` keyword is used before `LocalUnhealthyApps` (`Relocate All LocalUnhealthyApps`...), all elements of the application set will be moved to the first element of the `NonLocalSuitableNode` set. If the `All` keyword is used before both sets (... `To All NonLocalSuitableNode`), all elements of the application set will be moved to the elements of the node set according to the following scheme: the i-th element of the application set is migrated to the $(i \bmod |nodeset|)$-th element of the node set.

CPN/CPS-Specific Elements While it is possible to apply *Rango* in arbitrary scenarios, the language includes four features that are particularly useful in CPN/CPS development: CPN/CPS-specific *components* (violet), *attributes* (purple), *adaptation actions* (blue), and *scopes* (orange) (cf. Table 5.3[1]). Each of these features is demonstrated by the following code snippet:

```
DefineSet LeastImportantApps:
App System Where Importance Min

If SystemOverloaded
Then Stop LeastImportantApps
```

This code snippet specifies a rule that stops the least critical application in the CPS in case of an overload[2].

In the previous section queries to group several components dynamically into sets have already been introduced. To achieve the desired behavior, a set has to be defined that contains the least critical application(s). Therefore, *Rango* includes keywords that automatically address typical components of a CPS (applications, nodes, and communication channels). In the code snippet, the keyword `App` is used to consider all applications. Furthermore, attributes of the components such as their health are used for comparison or maximum/minimum determination. *Rango* offers 43 pre-defined CPN/CPS-specific attributes that are ready to use (cf. Table 5.3).

[1] Please refer to the Appendix A for the entire list of available CPN/CPS specifics.

[2] For readability reasons, the definition of the `SystemOverloaded` condition in the code snippet is omitted.

CPS-specific element	Keyword	Remark
Components	App	Applications such as brake control
	Node	Processing nodes such as ECUs
	Comm	Communication channels such as CAN
Attributes	Importance	Describes in $\mathbb{N}_0$ the criticality of an application
	Health	Describes in (-∞,1] whether a component meets its requirements
	Period	Period in which an application is executed
	Scheduling	Scheduling policy of a node or communication channel
	Capacity	Remaining computational power of a node or communication bandwidth of a communication channel
	...	38 further CPS-specific attributes
Actions	Stop	Terminates an application
	Relocate	Migrates an application to another node
	TunePeriod	Changes the period of an application
	SetPriority	Sets the priority of an application
	SetScheduling	Changes the scheduling policy of a node or communication channel
	...	14 further CPS-specific actions
Scopes	Local	All components local to a certain node
	System	All components in a certain CPS
	Global	All components in the entire CPN
	NonLocal	All components not local to a certain node
	NonSystem	All components not belonging to a certain CPS
	...	4 further scopes

Table 5.3: CPN/CPS specific components, attributes, adaptation actions and scopes included in *Rango*.

In the above code snippet, *Rango*'s CPN/CPS-specific attribute `Importance` is used which allows to consider the mixed-criticality (cf. Section 3.1) by making it possible to specify rules that realize a dynamic prioritization of critical tasks, e.g., in overload situations as in the above code snippet. In the provided example the prioritization is achieved by terminating (`Stop`) the least important and thus least critical application in the entire system. There might be other, even better, adaptation actions to resolve the overload as can be seen in Table 5.6 of Section 5.4. To provide a maximum of flexibility and opportunities for adaptation to be able to learn the best actions in different situations, *Rango* includes 19 keywords that represent adaptation actions in CPN/CPS such as migrating applications (`Relocate`). An overview of possible adaptation actions in relation to different conditions can be found in Table 5.6 of Section 5.4

Finally, different scopes are introduced in *Rango* due to the distributed nature of mixed-critical CPNs. In an example platooning [Les+21] scenario, several autonomous vehicles drive in a convoy with small inter-vehicle distances. Those vehicles need to coordinate their inter-vehicle gaps via IEEE 802.11p communication. The overarching CPN would therefore consist of multiple CPS (the cars) (cf. Section 2.1). Thus, each CPS within the CPN includes several processing nodes (i.e., ECUs), which each have their own feedback loop for decision making. In such systems-of-systems, scopes such as "in the current CPS", "on the current processing node", or "somewhere in the whole CPN" play an important role in how to intuitively describe the desired adaptation behavior. For instance, a rule may migrate all applications that are running on an ECU to another ECU in the same car and not to an arbitrary ECU in the whole CPN. *Rango* offers multiple scopes with corresponding keywords for the specification of such rules: (i) *Global* refers to all components in the entire CPN (e.g., all applications within the whole platooning scenario), (ii) *System* refers to all components of the corresponding CPS (e.g., all applications running in a single car) (iii) *Local* refers to all components on the same node as the feedback loop (e.g., all applications running on the same ECU). Based on these basic scopes, several composite scopes for the rules can be derived, e.g., *NonLocal* refers to all components that are *not* located on the local node, and *NonSystem* refers to all components *not* belonging to the CPS of the local nodes. For example, a component like an application is *Local* to an ECU if the functionality is executed on this ECU. All applications not running on this node can be referred to as *NonLocal* and all applications not belonging to the CPS can be referenced by *NonSystem*.

Structure of Rule Files *Rango* rules are stored in `.rul` files. This way, the initial rule set or updates of rules without knowledge on LCS or implementation of mixed-critical CPN can be written. As depicted in Figure 5.7, such files consist of two parts: the configuration part and the rule set. In the configuration part, various configuration directives can be defined. Thereby, settings for the rule evaluation times (e.g. `EvaluationPeriod`), action execution, learning parameters, and reward calculation parameters can be specified. An overview of the configuration directives is

provided in Table 5.4. The second part of the file — the rule set — consists of all sets, conditions, actions, and rules. Any sequence of sets, conditions, actions, and rules is valid as long as there are no forward references (single pass rule parser).

```
1  /* ConfigurationDirectives */
   EvaluationPeriod 0.1
   ---------------------------------------------------------------------------
2  /* Rule Set */
   /* Named Sets */
   DefineSet localUnhealtyApps: App Local Where Health < 0
   DefineSet leastImportantApps: App Global Where Importance Min

   /* Named Conditions */
   DefineCondition localUnhealtyAppsOnNode:
   Cardinal (localUnhealthyApps) > 0

   /* Named Actions */
   DefineAction stopLeastImportantApp:
   Stop [All] leastImportantApps

   /* Rules */
   If localUnhealtyAppsOnNode Then stopLeastImportantApp
```

Figure 5.7: Rule file structure .rul of *Rango*.

Furthermore, *Rango* improves traceability and explainability of learning as it allows to export rule sets into a human-readable format after the LCS performed online learning. Such modified rule sets are stored in `.rulx` files. Files in this format start with a comment that states which middleware/LCS instance modified the rule set. The remainder is structured in the same way as a `.rul` file (cf. Figure 5.7). However, each rule is now followed by the corresponding `Reward` and the `Experience`. The reward is a real number that expresses the benefit of the rule execution for the system goal (cf. Section 5.2.3) and the experience value indicates how often a rule was executed. Such a modified rule file in the `.rulx` format can also be re-parsed in the internal representation and, hence, be re-used as an initial rule set for adaptation. This makes it possible to benefit from previous learning, e.g., performed by another LCS in a similar environment.

Rule Processing Rules can basically be processed either time or event-driven. In the event-driven processing, the rule processing takes place exactly when an attribute value of a component (i.e. health, demand, ...) changes. In the case of time-driven evaluation, the rule set is processed periodically at fixed time intervals. A detailed analysis of the rule processing complexity for time and event driven evaluation can be found in the Section 7.1.3. In the conducted evaluation, the time-driven processing has been used. First, it supports a well-defined timing behavior. Second, it can be observed that the processing frequency necessary for time driven approach is usually lower than the change frequency of attribute values. Thus, the time-driven approach

	Keyword	Value	Default	Remark
Rule Evaluation Times	`EvaluationPeriod`	Real	0.1s	Period for evaluation of the rule set
	`RewardDelay`	Real	2.0s	Delay to update the reward after an action execution & next action can take place
Action Execution	`SkipImpossibleAction`	On/Off	On	Skips actions which cannot be executed (e.g. due to empty sets actions are skipped even if they have the best reward)
	`Lag`	Integer	1	Number of times the condition has to be consecutively true before the action is executed (e.g. to avoid race conditions)
Learning Parameters	`LearningRate`	Real	0.5	The learning rate
	`InterferenceWeight`	Real	0.5	The influence of interferences for the learning rate
	`MinLearningRateScale`	Real	0	The minimum learning rate scale (e.g. in case of interferences)
Reward/ Fitness Calculation	FitnessWeightAppHP	Real	0.7	The influence of application health/performance ratio for the application fitness
	`FitnessWeightAppImportance`	Real	0.5	The influence of application importance for the application fitness
	`FitnessWeightApp`	Real	0.9	The influence of the application fitness for the overall fitness
	`FitnessWeightNode`	Real	0.05	The influence of the node fitness for the overall fitness
	`FitnessWeightComm`	Real	0.05	The influence of the communication channel fitness for the overall fitness
	`FitnessLifeSignScale`	Real	1	The scale factor for life sign age limit to consider an application or node crashed in fitness calculation
	`FitnessCrashHealth`	Real	-1	The *health value* for a crashed application or node
	`FitnessSkicknessBoost`	Real Real	0.1 3	The boost for *health values* below a sickness threshold
	`FitnessScope`	Global/ System/ Local	Global	The scope for the fitness calculation
	`FitnessMode`	Integer	0	Fitness calculation mode (0 = health performance product version a, 1 = health performance product version b, 2 = health performance product version c, 3 = health performance sum)

Table 5.4: Configuration Directives of *Rango*

produces less load. Therefore, all rules are processed periodically with the definable `EvaluationPeriod` (cf. Table 5.4). Additionally, the time driven approach can be used to reduce the number of interferences in distributed LCS. Delaying an action of a LCS instance by a well defined amount of time if a competing action of another LCS instance has been detected can avoid interference of both actions, see Section 5.2.3.

A detailed analysis of *Rango*'s parsing complexity, rule processing complexity and rule set memory footprint can be found in Section 7.1.3. Additionally, a usefulness and usability study of *Rango* can be found in Sections 7.1.1 and 7.1.2.

5.2.3. Reward Calculation

In LCS, learning at run-time (online learning) is based on a so-called *reward* value. This numerical fractional value ($reward \in \mathbb{R}$) indicates how well or bad the execution of a rule action has changed the state of the system. Thereby, the rule processing can learn what are good and bad rules for the current condition. Positive values indicate an improvement of the system state while negative values indicate a degradation. As larger a positive value as more the state has improved. As lower a negative value, as more the state has degraded. To derive such a numerical reward value, the state of the system has to be quantified by a so-called $fitness$ value. This value rates the overall state of the system by a fractional number ($fitness \in \mathbb{R}$). Having such a fitness value, the reward of an action can be measured by calculating the difference between

the fitness after and before the action. Assume $fitness_{new}$ being the fitness after and $fitness_{old}$ being the fitness before an action, then the measured reward is given by:

$$reward_{measured} = fitness_{new} - fitness_{old} \tag{5.9}$$

The fitness can be calculated based on values retrieved from the monitoring component. How this is done in detail will be explained in the next subsections. However, the monitoring component needs time to observe the effect of an action. Therefore, there has to be a certain delay between initiating an action and retrieving the value for $fitness_{new}$ after the action. This delay is called *reward delay*. It ensures that the monitoring component is able to properly measure the outcome of an action on the system fitness. This delay can be specified in *Rango* globally and also individually for each action using the keyword `RewardDelay` (cf. Table 5.4).

To avoid trashing of reward values and do get a smooth learning curve, the measured reward is softened by a learning function. Let $reward_{old}$ by the current reward of an action and $LearningRate$ be a fractional number in the range of [0,1]. Then the new reward of an action is calculated from the measured reward by the following learning function often used for LCS:

$$reward_{new} = reward_{old} + learningRate \cdot (reward_{measured} - reward_{old}) \tag{5.10}$$

The learning rate determines, how fast the new reward is updated by the measured reward. Let's look at the two border cases: If the learning rate is 0, the new reward is identical to the old reward. So no learning takes place. If the learning rate is 1, the new reward is identical to the measured reward. So the old reward has no influence at all. Thus, the learning rate weights the influence of the history (old reward) on the new reward.

Reward Calculation in a Distributed LCS In a central LCS, only one action is executed at a time and then the result (reward) of this action is evaluated after the reward delay. In a distributed LCS, different local instances of the LCS can trigger actions independently of each other. If such an action is within the delay of another action, these actions interfere and the result (measured reward) can no longer be clearly attributed to one of these actions. This has to be taken into account when calculating the new reward in a distributed LCS.

In the following, the detection and handling of such interferences for the reward calculation is discussed in more detail. Afterwards, the detailed $fitness$ calculation is presented. Several of the parameters introduced in the next sections can be directly defined in the configuration section of a Rango rule file (cf. Table 5.4). These parameters are set in Courier font (e.g. `RewardDelay`).

Interference Detection and Handling To be able to detect interferences, actions are transmitted in the system via broadcast messages[1] (cf. Section 5.3). If such a broadcasted action message of a remote LCS instance is received during the reward delay of a local action execution, these two actions obviously overlap within the reward delay. Thus, an interference is detected. Broadcasting action messages also has the general advantage that each local instance of the LCS can pick and execute the local part of a global action (if e.g. the global action "*increase the priority of all applications with an importance higher a given threshold*" is broadcasted, each local LCS instance can apply this action to its affected applications). Furthermore, the broadcasted action messages can be used to reduce the interferences within the reward delay period. If an action message is received from another LCS instance, the next rule evaluation of the local LCS instance can be delayed so that it does not take place in the reward delay period of the received action. Due to communication delay and race conditions this approach cannot avoid all interferences, but it can significantly reduce their number[2].

If still an interference is detected, the learning rate for the reward calculation can be reduced to account for the fact that the result can no longer be unambiguously assigned. Thus, the learning rate can be scaled as follows:

$$learningRate = learningRateScaling \cdot \texttt{LearningRate} \tag{5.11}$$

Thereby, the unscaled basic `LearningRate` can be defined in the configuration directives of the rule file.

Now let's have a look on how *learningRateScaling* can be derived. When reducing (scaling) the learning rate, first the strength of the interference needs to be evaluated, thus how much time of the reward delay was affected by the interference. This is illustrated in Figure 5.8. The LCS instance of middleware MW_A has executed an

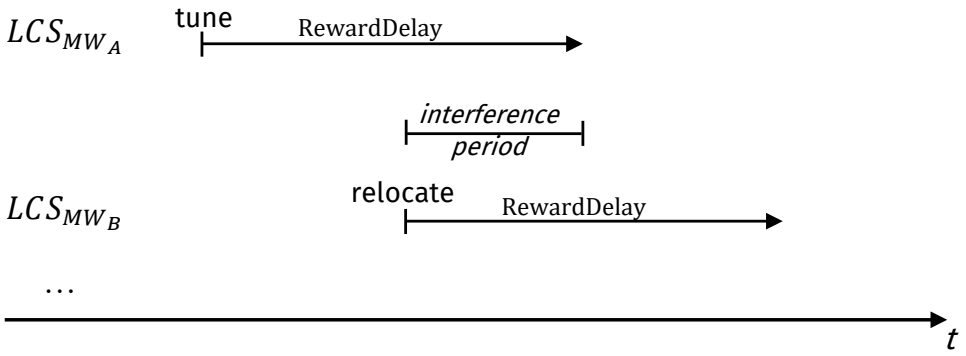


Figure 5.8: Example for an interference between the reward delays of two actions (`TunerPeriod`, `Relocate`) executed by distributed LCS instances of the middlewares MW_A and MW_B.

[1] The resulting overhead of the action notification broadcast is affordable, since adaptation actions are much rarer than the regular data exchange in the system (cf. Section 7.3.5).

[2] Complete avoidance of interferences is future work and will be discussed in Section 9.2.

action to tune the period of an application and now waits for the reward delay to pass. During waiting, an interference happens caused by the LCS instance of middleware MW_B executing an action to relocate an application. The marked *interference period* gives the time in which the reward of the middleware MW_A action is spoiled by the MW_B action. As smaller this interference period is, as smaller is the effect of the interference on the reward. Thus, an *interference strength* can be defined based on the interference period and the reward delay:

$$interferenceStrength = \frac{interferencePeriod}{\texttt{RewardDelay}} \tag{5.12}$$

Since the maximum possible length of the interference period is the reward delay, the interference strength is in the range of $[0, 1]$.

Furthermore, the number of interferences in the reward delay needs to be considered (the more interference's, the more results are mixed). For each interference j within the reward delay, a corresponding interference strength can be calculated accord to formula 5.12. In addition, it is useful to define an `InterferenceWeight` to specify the influence of interferences on the learning in general. Assuming we have n interferences in the reward delay of an action, the learning rate scaling for the `LearningRate` can be defined as follows:

$$learningRateScaling = 1 - \texttt{InterferenceWeight} \cdot \sum_{j=1}^{n} interferenceStrength_j \tag{5.13}$$

Furthermore, a lower limit `MinLearningScale` for $learningRateScaling$ can be defined to maintain a minimum rate of learning even in the presence of interferences:

$$\begin{aligned} &\text{if } learningRateScaling < \texttt{MinLearningScale} \Rightarrow \\ &\quad learningRateScaling = \texttt{MinLearningScale} \end{aligned} \tag{5.14}$$

Fitness In general, the fitness for reward calculation can be based on *health values*. In the following, a parameterized model for such a fitness calculation is introduced.

In this model, the fitness function is composed of a weighted combination of (i) applications fitness $fitness_{\sum_{\forall apps}}$, (ii) nodes fitness $fitness_{\sum_{\forall nodes}}$, and (iii) communication channels fitness $fitness_{\sum_{\forall comms}}$. Thereby, the fitness can be calculated using different scopes (`FitnessScope`). These scopes can be global[1], system[2] or local[3]. The weighting $[0, 1]$ of the subareas is specified by (i) `FitnessWeightApp`, (ii) `FitnessWeightNode`, and (iii) `FitnessWeightComm` (cf. Table 5.4).

[1] $\sum_{\forall apps}$ = all applications of the entire CPN, $\sum_{\forall nodes}$ = all nodes of the entire CPN, $\sum_{\forall comms}$ = all communication channels of the entire CPN

[2] $\sum_{\forall apps}$ = all applications of the local CPS, $\sum_{\forall nodes}$ = all nodes of the local CPS, $\sum_{\forall comms}$ = all communication channels of the local CPS

[3] $\sum_{\forall apps}$ = all applications of the local node, $\sum_{\forall nodes}$ = the local node itself, $\sum_{\forall comms}$ = all communication channels of the local node

$$fitness = \frac{\begin{array}{c}(\texttt{FitnessWeightApp} \cdot fitness_{\sum_{\forall apps}} + \\ \texttt{FitnessWeightNode} \cdot fitness_{\sum_{\forall nodes}} + \\ \texttt{FitnessWeightComm} \cdot fitness_{\sum_{\forall comms}})\end{array}}{\begin{array}{c}(\texttt{FitnessWeightApp} + \texttt{FitnessWeightNode} + \\ \texttt{FitnessWeightComm})\end{array}} \tag{5.15}$$

Application Fitness The application fitness function is mainly based on the health of an application. However, there is another important parameter which influences this function: the performance of an application, which can be lowered by tuning (up to pausing). Lowering the performance of applications can improve the health of the system, but thereby the performance of the entire system is reduced. The same is true vice versa. Thus, health and performance may be competing optimization goals.

To handle this, a $performance$ value $[0, 1]$ is introduced, which indicates whether an application is working at its nominal performance (1) or how far this performance has been reduced by tuning (< 1). A value below 0 states that the application has been paused completely. In general, four aspect of an application can be tuned: (i) priority, (ii) deadline, (iii) reaction time, and (iv) period. In case of priority tuning, the performance ($performance_{priority}$) is lowered if the priority of the application is lowered. Otherwise, is regarded to stay at 1. Let $tune_{priority}$ be the tuning factor of the application's priority with $priority_{tuned} = tune_{priority} \cdot priority$. Then, the performance regarding priority tuning can be expressed by:

$$performance_{priority} = \begin{cases} 1 & \text{if } tune_{priority} \geq 1 \\ tune_{priority} & \text{if } tune_{priority} < 1 \end{cases} \tag{5.16}$$

The performance of the system is also lowered if the deadline of an application is extended. Otherwise, the performance is regarded to stay at 1. Let $tune_{deadline}$ be the tuning factor of the application's deadline with $deadline_{tuned} = tune_{deadline} \cdot deadline$. Then, the performance regarding deadline tuning can be expressed by:

$$performance_{deadline} = \begin{cases} 1 & \text{if } tune_{deadline} \leq 1 \\ \frac{1}{tune_{deadline}} & \text{if } tune_{deadline} > 1 \end{cases} \tag{5.17}$$

The same holds true for the reaction time. Let $tune_{reactiontime}$ be the tuning factor of the application's reaction time with $reactiontime_{tuned} = tune_{reactiontime} \cdot reactiontime$. Then, the performance regarding reaction time tuning can be expressed by:

$$performance_{reactiontime} = \begin{cases} 1 & \text{if } tune_{reactiontime} \leq 1 \\ \frac{1}{tune_{reactiontime}} & \text{if } tune_{reactiontime} > 1 \end{cases} \tag{5.18}$$

The tuning of the period has similar effects on the performance than the tuning of

the deadline or reaction time. The performance is lowered if the period is increased and is regarded to stay at 1 if the period is shortened or stays the same. Let $tune_{period}$ be the tuning factor of the application's period with $period_{tuned} = tune_{period} \cdot period$. As a special case here, a tuning factor below zero ($tune_{period} < 0$) indicates that an application is paused ($period_{tuned} = \infty$) and thus the performance is zero. Then, the performance regarding period tuning can be expressed by:

$$performance_{period} = \begin{cases} 0 & \text{if } tune_{period} < 0 \\ 1 & \text{if } 0 \leq tune_{period} \leq 1 \\ \frac{1}{tune_{period}} & \text{if } tune_{period} > 1 \end{cases} \tag{5.19}$$

Since the overall performance of an application is already lowered if one of these partial performances is lowered, the overall performance is given by the minimum of the partial performances:

$$\begin{aligned} performance = \min(&performance_{priority}, performance_{deadline}, \\ &performance_{reactiontime}, performance_{period}) \end{aligned} \tag{5.20}$$

The range of $performance$ according to the above equation is [0,1], because all partial performances are in that range and therefore the minimum is as well.

A major question is now how to combine health and performance to calculate the fitness of an application? Increasing health or performance has to have an increasing impact while decreasing health or performance has to have a decreasing impact on the fitness. In the following two different approaches are proposed: *gradient based combination* and *offset based combination*:

Gradient based combination: Here, the $performance$ value represents the gradient of the $fitness_{app}(health_{app})$ function as shown in Figure 5.9. If $performance = 1$, the fitness is directly given by the health. A $performance < 1$ flattens the slope. This leads to the desirable behavior, that the reduction of the performance of an application with tuning (to resolve an overload situation) as well as the restoring of the performance by diminishing the tuning (if there is no overload situation anymore) can lead to an increase of the system fitness and thus a positive reward. Figure 5.9 demonstrates this and explains it in the caption.

To get more detailed, there are different options for this gradient based approach, which differ in the way negative *health values* are treated (cf. Figure 5.10).

(a) positive and negative *health values* are treated the same way, as shown by the red line marked a: in Figure 5.10 and given by the following equation:

$$fitness_{app} = health_{app} \cdot performance \tag{5.21}$$

Thus, the reduction of the performance of unhealthy applications increases the

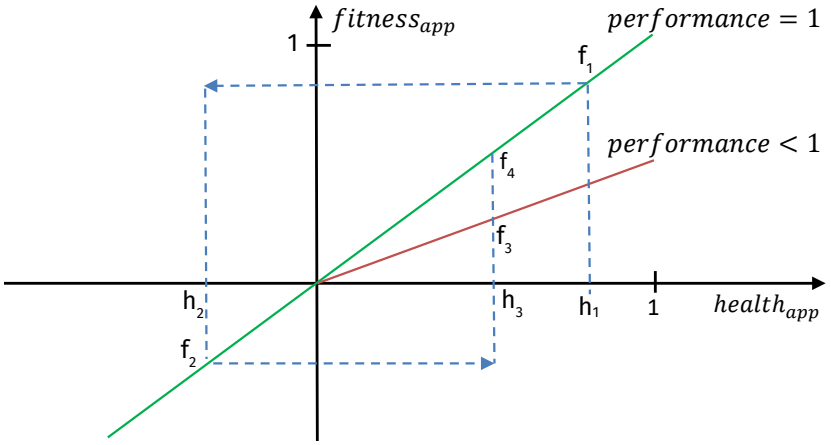


Figure 5.9: Gradient based combination of health and performance:
The example sequence given by the blue arrows shows that tuning in an overload situation generates a fitness gain if it restores the system to health. Undoing the tuning when the overload situation no longer exists also generates a fitness gain. Let us assume the application is initially healthy ($health_{app} = h_1$) and untuned ($performance = 1$). This results in fitness f_1. Now an overload situation makes the system unhealthy ($health_{app} = h_2$). This results in the reduced fitness f_2. The adaptation mechanism now counteracts the unhealthy state by tuning ($performance < 1$), which leads to the improved health h_3 and fitness f_3. Therefore, this tuning action gets the positive reward $f_3 - f_2 > 0$. After another while, the reason of the overload does no longer exist and the adaptation mechanism reacts by undoing the tuning ($performance = 1$). This leads to fitness f_4. Therefore, this undoing action also gets a positive reward $f_4 - f_3 > 0$.

fitness (as more resources are left for other (healthy) applications)

(b) the fitness is not influenced by the performance if the application is unhealthy, as shown by the green line below 0 marked b: in Figure 5.10 and given by the following equation:

$$fitness_{app} = \begin{cases} health_{app} \cdot performance & \text{if } health_{app} \geq 0 \\ health_{app} & \text{if } health_{app} < 0 \end{cases} \tag{5.22}$$

(c) the reduction of the performance of unhealty applications decreases the fitness by the same amount as it does for healthy applications, as shown by the blue line below 0 marked c: in Figure 5.10 and given by the following equation:

$$fitness_{app} = \begin{cases} health_{app} \cdot performance & \text{if } health_{app} \geq 0 \\ health_{app} \cdot (2 - performance) & \text{if } health_{app} < 0 \end{cases} \tag{5.23}$$

To specify the extent to which the performance has an influence on the fitness

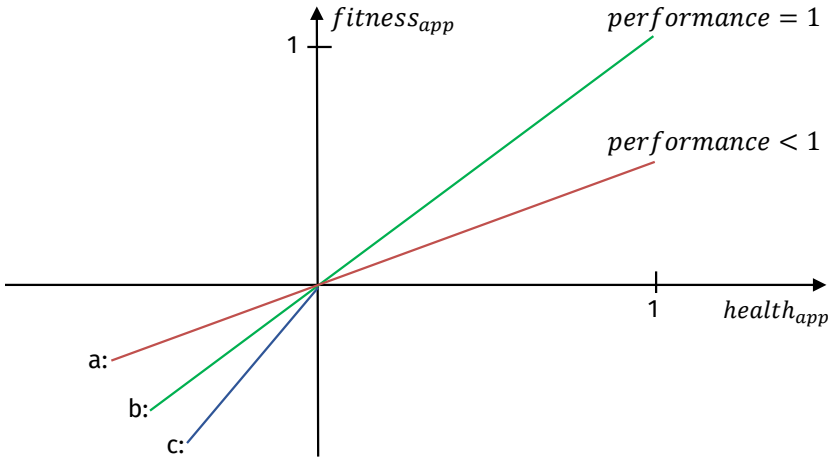


Figure 5.10: Gradiant based combination: options for negative *health values*.

of the application, an auxiliary weighting factor `FitnessWeightAppHP` $[0, 1]$ can be defined (cf. Table 5.4). The *performance* in the application fitness calculation (Equations 5.21-5.23) is then replaced by the $weightedPerformance$ which can be defined as follows:

$$weightedPerformance = \texttt{FitnessWeightAppHP} \cdot performance + (1 - \texttt{FitnessWeightAppHP}) \tag{5.24}$$

Offset based combination: Alternatively, a linear mapping of the two quantities health and performance to the fitness can be achieve using the sum of both. So

performance and health simply define an offset to each other when calculating the fitness. Again, the weighting factor `FitnessWeightAppHP` $[0, 1]$ can be used to shift the influence of the two sub-aspect on the fitness. This is expressed by the following equation:

$$\begin{aligned} fitness_{app} = \texttt{FitnessWeightAppHP} \cdot health_{app} + \\ (1 - \texttt{FitnessWeightAppHP}) \cdot performance \end{aligned} \tag{5.25}$$

If `FitnessWeightAppHP` $= 1$, only the health determines the fitness. In the other extreme, if `FitnessWeightAppHP` $= 0$ only the performance determines the fitness. For any other value in between, health and performance are added as a weighted offset.

The two combination modes including the sub-variants of the gradiant based mode can be controlled by the configuration directive `FitnessMode` in the configuration section of the rule file, cf. Table 5.4).

Finally, the *importance* of the applications can also be considered when calculating the overall applications fitness $fitness_{\sum_{\forall apps}}$. A factor `FitnessWeightImportance` can be used to specify the extent to which the applications *importance* has an influence on the overall application fitness (cf. Table 5.4).

$$\begin{aligned} fitness_{\sum_{\forall apps}} = \texttt{FitnessWeightImportance} \cdot \frac{\sum_{\forall apps} fitness_{app} \cdot importance_{app}}{\sum_{\forall apps} importance_{app}} + \\ (1 - \texttt{FitnessWeightImportance}) \cdot \frac{\sum_{\forall apps} fitness_{apps}}{|apps|} \end{aligned} \tag{5.26}$$

The above formula can be easily explained when first looking at the two extremes for `FitnessWeightImportance`. If `FitnessWeightImportance` $= 0$, the *importance* has no influence in the equation. The overall application fitness simply calculates to the average of the fitness for all applications (second line of the equation). If `FitnessWeightImportance` $= 1$, the fitness of an application is weighted by its *importance* when calculating the average (first line of the equation). For any `FitnessWeightImportance` value in between, a mix of both extremes can be configured.

Node and Communication Channel Fitness Since tuning and importance applies to applications, the fitness calculation for nodes and communication channels is much simpler. Here, the node and communication channel fitness can be derived directly by building the average of their *health values*.

$$fitness_{\sum_{\forall nodes}} = \frac{\sum_{\forall nodes} health_{node}}{|nodes|} \tag{5.27}$$

$$fitness_{\sum_{\forall comms}} = \frac{\sum_{\forall comms} health_{comms}}{|comms|} \tag{5.28}$$

Reward Boosts There are two further options to influence the fitness calculation via *health value* modification. This allows to increase the reward for desired adaptation actions.

First, the influence of poor *health values* to the fitness can be reinforced by defining a sickness *boost* for *health values* below a sickness *threshold*. All *health values* lower than the threshold are then further decreased by the boost factor[1].

$$health^{boosted} = \begin{cases} health & \text{if } health \geq threshold \\ boost \cdot (health - threshold) + threshold & \text{if } health < threshold \end{cases} \tag{5.29}$$

The principle of this idea is illustrated in Figure 5.11.

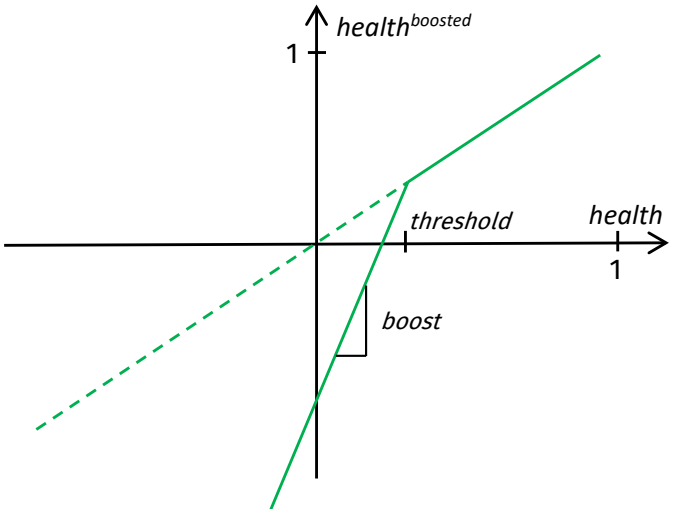


Figure 5.11: Reinforce the influence of poor *health values* to the fitness by defining a sickness *boost* for *health values* below a sickness *threshold*.

As larger the boost value is chosen, as more reward is gained for bringing a component back to a healthy state. This is due to the fact, that the reward of an action is calculated by the fitness gain (cf. Equation 5.9). Now since the health is a vital part of the fitness,

[1]The values for the boost and for the threshold can be specified in the configuration directives of the rule file using the `FitnessSicknessBoost` *threshold boost* statement.

as lower the *health value* for a sick component is, as bigger is the gain for making the component healthy again.

Second, system components may fail silently (e.g. by an application or node crash). Hence, the component's *health value* stays on the last value retrieved before the failure. However, the silent failures can be detected by the middleware and the rule engine/set based on missing life signs. Thus, it is reasonable to assign a negative *health value* to such components if the maximum life sign age is exceeded to enable a positive reward when the failed components are restored. This can be done by the parameter `FitnessCrashHealth`, which defines the *health value* of a component with expired life sign (cf. Section 5.1). As a default, the expiration age has been selected as twice the life sign period (cf. Table 5.2, update period) of a component . By the parameter `FitnessLifesignScale`, this period can be scaled and thus shortened or extended.

5.3. Execute

The *Execute* stage as the final stage of the MAPE-K loop is responsible to perform a planned adaptation action. A selected action is broadcasted (cf. Section 5.2.3) in the CPN and each local instance of the middleware performs its local part. Thereby, the actions are SIMD (Single Instruction Multiple Data) actions. An action is performed simultaneously for multiple components (applications, nodes, communication channels) with different data. An simple example is tuning application periods relative to a current value as shown in the *Rango* snippet below (current value is multiplied by 0.2). Thereby, each application in the overall CPN which meets the application query clause (`Health < 0.5`) can receive a different resulting tuning value. Like the conditions, actions can have different scopes. In the example below, a global scope is used for the action.

```
TunePeriod All App Global Where
Health < 0.5 to Current * 0.2
```

Therefore, different scope combinations for conditions and actions can be used to build a rule set. In general, the condition and action can have either the same or different scopes. A purely local rule set (local conditions and actions) leads to the least interferences but has the disadvantage of no global view, thus no global optimization goal can be found (e.g. the least important app within the overall system can't be found). A purely global rule set (global conditions and actions) on the other hand guarantees best global view but leads to the most interferences. As an alternative, a combination of different scopes for actions and conditions is possible. On the one hand, e.g. local conditions and global actions can be used. This results in only few to medium interferences and benefits from the usually higher resolution of the local monitoring data. On the other hand, the conditions can be global and the actions local. This still leads to only a few interferences but has to deal with the lower resolution of the remote monitoring. Table 5.5 compares the interferences, views and impacts of different scope combinations for conditions and action, ranging from local middleware scope (local), system wide CPS scope (system) to global CPN scope (global).

condition scope	**action scope**		
	local	*system*	*global*
*local**	fewest interferences, local view, local impact	few interferences, local view, system impact	few-medium interferences, local view, global impact
system	few interferences, system view, local impact	medium interferences, system view, system impact	medium-many interferences, system view, global impact
global	few-medium interferences, global view, local impact	many interferences, global view, system impact	most interferences, global view, global impact

Table 5.5: Impact of different scope combinations for conditions and action on interferences.
(*Local conditions also benefit from the usually higher local monitoring resolution compared to remote monitoring.)

The combination of different scopes enables to freely select between purely local, purely global and hybrid self-adaptation schemes. Therefore, this local versus global paradigm can combine the advantages of both worlds (e.g. local = no single point of failure; global = global optimization goals, ...). To the best of the author's knowledge, this is not yet accomplished in any other approach.

5.4. Summary

In this Section 5, the designed adaptation mechanism of the middleware *Chameleon* has been presented which enables an efficient and flexible management of mixed-critical requirements. Thereby, each stage of the used MAPE-K loop architecture has been explained in detail. The sampling of various monitoring data and derivation of health values are used to determine the components and overall system condition. The further transformation of the highly specific monitoring data into abstract health values provides a unified concept for the definition of the acceptance and target space. It is used by the LCS which realizes the analyzing and planning component as well as the knowledge base. In *Rango*, the health values can be used in the conditions to trigger adaptation. Thereby, *Chameleon* offers many adaptation actions to keep the system in the acceptance or target space (proactive or reactive adaptation). Besides the health values, the detailed monitoring data can then be used in *Rango* to deduct the best potential adaptation action for the current condition. The action which is expected to lead to the highest reward is then executed.

Table 5.6 exemplarily shows how the health values can be used to derive the compo-

health < limit	potential reason	useful additional health & monitoring data	potential adaptation							
			(re-) start app	stop app	re-locate app	remove app	tune app	com-press data	change sche-duling node	change sche-duling comm. channel
appli-cation (app) health	node: overload, performance loss, failure	node		x	x	x	x		x	
	comm. channel: overload, performance loss, failure	comm.		x	x	x	x	x		x
	design flaw	node & comm.							x	x
	app: failure	-	x							
node health	overload, performance loss, failure	application		x	x	x	x		x	
comm. channel health	overload, performance loss, failure	node		x	x	x	x	x		x

Table 5.6: Example deduction of potential adaptation measures (marked with "x") based on health value violation including additional health and monitoring information to distinguish the reasons for the violation.

nents and system condition and which potential adaptation measures can be deducted. The first column describes which health value violation (target or acceptance space) triggers the adaptation. The second column lists several reasons for the violation. Column three outlines different additional health or monitoring information to distinguish these reasons. The last column shows potential adaptation measures (actions) to improve the components and system state under these conditions. For example, if the local application health in combination with the local node health are lower than the defined limit but the communication is still fine, a node overload might be the source of the problem. Hence, a relocation of the application might be a good adaptation action. To find the best suitable processing node for relocation, the node capacity sampled from monitoring can additionally be used as input value. In an automotive example, the navigation might e.g. be moved to another computing node to maintain the timing constraints of more important functions like braking while keeping also the navigation operational (as used e.g. in the *Rango* example in Figure 5.6).

6. Implementation

The *Chameleon* middleware has been implemented in C++. For practical evaluation, *Chameleon* is embedded in a simulated environment as illustrated in the extract of the system model (cf. Figure 3.1) in Figure 6.1. The simulated parts are colored in gray. In comparison to a real-world implementation, the simulated environment offers the opportunity to deeply look inside the adaptation processes and deficiencies will not cause physical harm. It also offers more flexibility in terms of multiple application scenarios and the size of the simulated environment. Thereby, *Chameleon* is completely independent of the simulated environment.

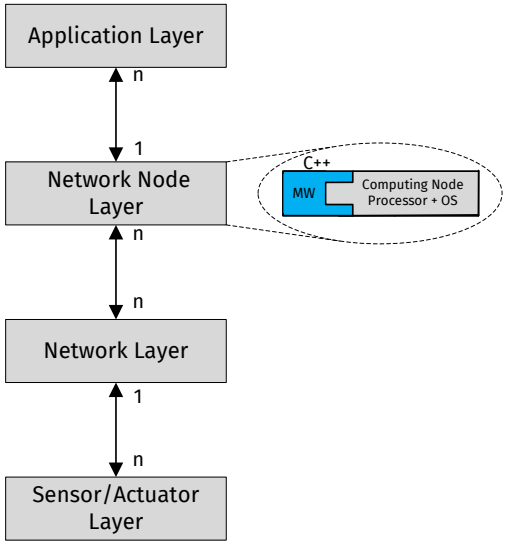


Figure 6.1: System Model with simulated environment (gray).

In the following, more details on the middleware architecture implementation (Section 6.1) and the simulated environment (Section 6.2) are briefly provided.

6.1. *Chameleon* Middleware Implementation

For the realization of *Chameleon*, the modules of the middleware architecture shown in Figure 4.1 have been prototypically implemented in C++ 11. The LCS-based MAPE-K Adaptation Logic has been implemented as introduced in Section 5 and consists of the modules shown in Figure 5.5. Like mentioned in Section 4, the middleware is implemented as a framework on which various adaptation mechanism can be built and

thus evaluated in the simulated environment. With few interface modifications, it can be used and tested also in a real-world environment.

Table 6.1 summarizes the Lines of Codes and Code Size of all *Chameleon* modules. The modules were compiled using *clang* version 5.0.1, target x86-w64-windows-gnu. The basic middleware itself is rather small with an overall code size far less than 100kBytes. Also the adaptation logic doesn't occupy much code space. The biggest module is the Rule Parser and Rule Writer with 93kBytes. However, this module may not necessarily be included in each *Chameleon* instance. On *Network Computing Node*s with low memory resources, the rule set can be compiled offline and loaded directly into the local Knowledge Base in binary format. Also, for the Rule Writer, the rule set modified by learning can be exported in binary form and written back in human readable form offline.

	Lines of Code	Code Size [kBytes]
Basic Middleware:		
Interfaces	716	22
Local Map	832	25
Load Handler	616	17
Request Handler	176	2
Total:	2340	66
Adaptation Logic:		
Monitor	1170	24
Analyze and Plan	1940	56
Execute	720	19
Rule Parser and Rule Writer	3835	93
Total:	7665	192

Table 6.1: Lines of Codes and Code Size of *Chameleon* modules.

6.2. Simulated Environment

To simulate the environment, OMNet++ (Objective Modular Network Testbed in C++) [OMN23] has been used. OMNet++ is a commonly known and well reputed event-based simulator for network processing. It offers many features and pre-defined modules to precisely simulate communication networks and sensor/actuator systems. For this thesis, it has been extended by modules to simulate realistic applications (cf. Section 6.2.1) and dynamics at runtime (cf. Section 6.2.2). Additionally, existing OMNeT++ modules have been adapted to simulate the *Network Computing Node*s. Thus, the developed simulated environment offers a basic platform for middleware research in the field of mixed-critical CPSs and mixed-critical CPNs. It is designed to efficiently determine, implement, calibrate and evaluate the compliance to application requirements in such systems.

The middleware implementation mentioned in the section above has been integrated into this simulation environment. Hence, a monolithic approach is used for the simulation. This has the benefit of better efficiency and timing behavior compared to co-simulators. Especially for simulating real-time environments, which is a key feature for MC-CPN, the monolithic approach is advantageous. In co-simulators, the coordination between the single simulators is highly complex and mostly comes at the cost of latencies in the system.

In the following, the major extensions to the OMNeT++ simulator are described in more detail.

6.2.1. Simulation of the Application

To simulate a realistic behavior of the application program execution, the processing of instructions on the computing nodes including all middleware interface functions as introduced in Section 4.1 is modeled. Additionally, all application parameters as noted in Table 3.1 and introduced in Section 3 are supported.

The application program is given by an XML file. Thereby, the program processing is inspired by petri nets [Inf23]:

Petri nets are a mathematical modeling tool to represent and analyze the behavior of systems with many interacting and concurrent components. They consist of places, transitions, and directed arcs that connect them. Thereby, places represent the system's states or conditions, transitions represent the actions or events that can occur in the system, and the arcs represent the flow of tokens (the system's resources such as messages) between places and transitions. Thus, an application program consists of two kinds of instructions:

Active instructions (transitions) which trigger an action and passive instructions (places) which wait for an event (cf. Table 6.2). An application program is defined by a sequence of such active and passive instructions. This sequence (also called a cycle) is executed 1 to n times with the given period parameter (cf. Table 3.1), where n can be in between 1 to infinite. The period can also be set to 0 for immediate repetition. Within a sequence, just as in petri nets, active instructions are not triggered until all previous reactive instructions have been completed. If a sequence of several instructions of the same kind occurs, the behavior depends on the instruction kind: In case of active instructions all events are triggered sequentially. If the sequence consists of passive instructions the application simultaneously waits for all events within the sequence to complete or timeout. Thereby, each passive instruction has its own timeout as they are activated and run simultaneously (and not sequentially). As a result, the largest timeout determines the maximum delay of the sequence. With this concept, polling and callback mechanism can be modeled. In order to implement asynchronous communication in addition to synchronous communication, a buffer of variable size is introduced. This buffer is used to store events that arrive at an application, but the application is not (yet) waiting for the event (e.g. if the event does not refer to the current sequence of passive instructions).

An application program XML file is structured as follows:

```
<Instructions>
  <InstructionTag attribute1=" " attribute2=" " .../>
    ...
</Instructions
```

The program is contained in the root element `Instructions`. Inside this element, different self-closing instruction tags with various attributes can be defined. In Table 6.2 the possible instructions and their tags are described for active and passive instructions. They cover all middlware interface functions described in Section 4.1 plus an instruction pair for local processing (*DoLocalProcessing*/ *WaitForLocalProcessing*). Table 6.3 displays the major instruction attributes.

`InstructionTag`	**Remark**
Active:	
`DoLocalProcessing`	Performs instructions on the local computing node.
`DoProcedureCall`	Calls a procedure of a local or remote application.
`DoProcedureReturn`	Returns the result of a local or remote procedure call.
`DoMethodInvocation`	Calls a method of a local or remote application.
`DoMethodReturn`	Returns the result of a local or remote method invocation.
`DoMessagePassing`	Passes a message to a local or remote application.
`DoMessageReturn`	Returns the result of a passed message.
`DoDataStream`	Initiates a datastream.
`DoRequestSensorData`	Request data of a sensor.
`DoDataToActuator`	Sends data to an actuator.
Passive:	
`WaitForLocalProcessing`	Wait for the completion of local processing.
`WaitForProcedureCall`	Wait for a procedure call.
`WaitForProcedureReturn`	Wait for the result of a procedure call.
`WaitForMethodInvocation`	Wait for a method invocation.
`WaitForMethodReturn`	Wait for the result of a method invocation.
`WaitForMessagePassed`	Wait for a message.
`WaitForMessageReturn`	Wait for the result of a passed message.
`WaitForDataStreamPortion`	Wait for datastream portions.
`WaitForSensorData`	Wait for sensor data.

Table 6.2: Active and passive `InstructionTags`.

Attribute	Remark
`Target`	Target (active) or source (passive) of the event.
`InstructionCount`	Number of instructions to be executed for this application event.
`Datasize`	Data size associated with the event.
`Timeout`	Timeout for a passive event.
`PreCondition`	Specifies a condition under which an instruction is executed.
`SkipCycles`	Executes the instruction only each nth cycle.
...	

Table 6.3: Major `InstructionTag` attributes.

As an example, the XML Code for a very simple application illustrated in Figure 6.2 is given:

```
<Instructions>
  <DoRequestSensorData Target="SensorX"
        InstructionCount="10000"/>
  <WaitForSensorData Target="SensorX"/>
  <DoLocalProcessing InstructionCount="120000"/>
  <WaitForLocalProcessing/>
  <DoProcedureCall Target="ApplicationY" InstructionCount="50"
        Datasize="256"/>
  <WaitForProcedureReturn Timeout="100"/>
  <DoDataToActuator Target="ActuatorZ" InstructionCount="10000"
        Datasize="128"/>
</Instructions>
```

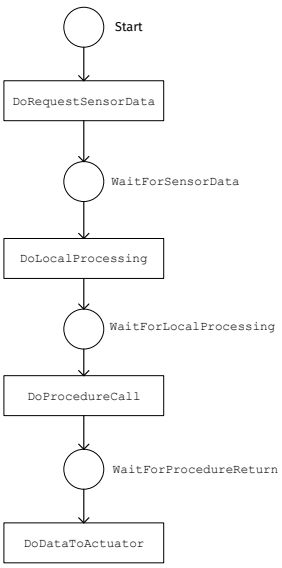


Figure 6.2: Example application program illustrated as petri net.

First, sensor data is requested. Its duration takes the execution of 10000 local computing node instructions to perform this request. After the data is received, local processing of 120000 computing node instructions is initiated. Then, a procedure of another application in the system is called and the result is expected. The data size of this call is 256 bits and it takes 50 local computing node instructions to perform the call. Thereby, a maximum wait time for the result of 100ms is defined. Finally, data is sent to an actuator of the system with a size of 128 bits which causes the execution of 10000 local computing node instructions.

An integer command parameter can be used to start (`command.i` = 1), stop (`command.i` = 2) or abort[1] (`command.i` = 3) the execution of an application. Also, an auto start option can be set, which starts the execution as soon as an application is instantiated. Using these options, flexible periodic (first instruction is an active instruction, period > 0) or reactive (first instruction is a passive instruction, period = 0) applications can be simulated. By using the `PreCondition` and `SkipCycles` attributes, conditional instruction execution and varying sequence durations can be introduced.

6.2.2. Runtime Control

The Runtime Control is a substantial extension to the runtime dynamics capabilities of OMNeT++. In particular, the change of the network topology as well as the insertion, removement and reconnection of modules at runtime has only limited support in OMNeT++. It can be only handled by explicit C++ programming. Therefore, a high-level Runtime Control module is introduced which allows a very comfortable high-level mechanism to define runtime events in the simulation environment. Thereby, the entire control is defined in an XML-file. A Runtime Control XML-file is structured as follows:

```
<Commands>
  <CommandTag attribute1=" " attribute2=" " .../>
   ...
</Commands>
```

All the Runtime Control events are contained in the root tag `Commands`. The events to be executed are defined in the subtags. The required information and parameters are passed via the attributes in the subtag. Table 6.4 gives an overview of the major available event `CommandTag`s. Some of the available attributes for those tags are further shown in Table 6.5.

[1] Stop ends the execution after finishing the current cycle while abort ends the execution immediately.

CommandTag	Remark
DeleteModule	Deletes a module.
CreateModule	Creates a module.
ConnectModule	Connects two modules.
DisconnectModule	Disconnects two modules.
SetParameter	Sets module parameters.
...	

Table 6.4: Major CommandTags.

Attribute	Remark
Name	Name of the module.
Class	Class of the module.
ToModule	Module to which a connection should be created.
FromModule	Module from which a disconnect should be realized.
Time [s]	Specifies the time (absolute or relative) at which an event is to be executed.
Jitter [s]	Delay of an event given by a random number between 0 and jitter.
Various module parameters	Module parameters which should be set.
...	

Table 6.5: Major CommandTag attributes.

As an example, the XML-Code of some simple runtime control events are given:

```
<Commands>
  <!-- crash NodeB at time 5 -->
  <DeleteModule Name="NodeB" Time="5"/>

  <!-- slow down NodeA due to thermal issues from 10 Mips
    to 7.5 Mips at time 10 -->
  <SetParameter Name="NodeA.paramCtrl"
    processingrate.i="7500000" Time="10"/>

  <!-- start ApplicationX at time 25 -->
  <SetParameter Name="ApplicationX" command.i="1" Time="25"/>
</Commands>
```

After 5 seconds a crash of the computing $Node_B$ is simulated by the deleting the corresponding module. At 10 seconds, a slow down of $Node_A$ to 7.5 Mips is triggered by accessing the parameter control (paramCtrl) module of the node and sending the new processing rate given by an integer value (`.i`). Finally, at 25 seconds ApplicationX is started by sending the start command (`comand.i = "1"`, cf. Section 6.2.1) to the application.

Using the runtime control module offers a flexible way to inject events during evaluation experiments.

7. Evaluation

To evaluate *Chameleon*, an extensive evaluation is conducted and presented in the following. First the usefulness, usability and performance of the rule language *Rango* is evaluated in Section 7.1. Afterwards, the real-time behavior of *Chameleon* is examined in Section 7.2. Finally, the usability of the rule-based MAPE-K adaptation mechanism for managing MC-CPN's and their complexity while considering the mixed-criticality is investigated in Section 7.3 by an automotive application scenario. Therefore, *Chameleon* has been embedded in a simulated environment (cf. Section 6) and nine evaluation scenarios are conducted which focus on various aspects.

7.1. *Rango* – Rule Language Evaluation

The rule language *Rango* (cf. 5.2.2) is evaluated threefold. First, five LCS experts are consulted to evaluate the usefulness of *Rango*, i.e., whether *Rango* has a practical worth (Section 7.1.1). Second, *Rango*'s usability, i.e., whether it is easy to understand and to write rules using *Rango* without extensive training, is evaluated in a study with 37 participants, mostly without experience in LCS (Section 7.1.2). Third, memory and computational overhead of *Rango* is assessed (Section 7.1.3).

7.1.1. Usefulness Study

Using *Rango* should provide a practical worth to potential users. Thus, its usefulness is assessed. Thereby, *Rango* supports potential users in two ways: (i) they can express an initial rule set for an LCS with the language and (ii) they receive human-readable output after learning that helps to trace and explain the learning process. Thus, the usefulness of *Rango* to writing an LCS rule set and for understanding and interpreting LCS output data is evaluated in the following.

Procedure & Methodology To evaluate *Rango*'s usefulness, an online questionnaire[1] for LCS experts, which consists of three parts, was designed. In the first part, the experts had to describe the status quo, e.g., which typical problems they face while using LCS or how they usually formulate rule sets. In addition, they had to rate how difficult it is to define an initial LCS rule set for (i) experts like themselves and (ii) system designers or administrators working in the industry on a 5-point Likert scale from 1 ("very hard") to 5 ("very easy"). Similarly, they had to rate how difficult it is to understand and interpret typical LCS output for both groups on the same scale. In

[1] The whole questionnaire is available at https://forms.gle/fgQK9cfcoqgM3XYYA.

the second part, *Rango* is introduced to the experts with a textual description similar to Section 5.2.2. In the third part, the experts again had to rate how difficult it is to define a rule set for themselves and system administrators but this time assuming that they could use *Rango*. The same applies to understanding and interpreting LCS output. The experts were additionally asked to provide feedback on *Rango* including its syntax, missing features, and — most importantly — its overall usefulness ("Would you use the language if it would be available?").

Participants Snowball sampling was applied to acquire participants. Therefore several LCS experts are contacted and asked to recommend further participants. Five LCS experts completed the questionnaire. On a scale from 1 ("not familiar") to 5 ("very familiar"), the self-rated average experience with LCS among the participants is 4.6. Four out of five participants have additionally used LCS to make a system adaptive.

Results In the first part of the questionnaire — before introducing *Rango* — three out of the five LCS experts have mentioned "rule encoding" or "finding good initial rules" as typical problems while working with LCS, which matches the motivation for developing the rule language *Rango*. While the quality of the rules is in theory independent from its formal representation, *Rango* can help to find better initial rules as these rules can be expressed easier in a more intuitive format.

Two experts mentioned that they usually formulate the initial rule set in bit strings. Improving this cumbersome way to specify rules is the core idea behind *Rango*. Another two experts start the learning process with an empty rule set and rely on automated, random generation. Consequently, such an approach might lead to rules that neither can be interpreted by humans nor it is traceable how the system found the rules. Especially in mixed-critical CPS/CPN use cases, which are typically complex, defining an initial rule set with low effort — as possible with *Rango* — is helpful to accelerate the learning process.

In the questionnaire, the LCS experts rated the easiness of writing a rule set and understanding the LCS output, both with and without *Rango*. They also estimated the easiness if system administrators had to write rules or understand the output. The results are summarized in Table 7.1 and provide three insights. First, *Rango* facilitates

Task	Group	Status quo	With Rango
Writing rules	LCS Experts	3.2	4.4
	System administrators	2.8	3.4
Understanding output	LCS Experts	3.4	4.0
	System administrators	2.2	3.6

Table 7.1: Average easiness of (i) writing rules and (ii) understanding LCS output on a scale from 1 ("very hard") to 5 ("very easy"), once with and once without *Rango*

the process of writing an initial rule set and understanding the LCS output for both

LCS experts and system administrators. Second, as far as writing rules is concerned, especially LCS experts benefit from *Rango*. The experts rate the easiness of writing rules with *Rango* 4.4 out of 5 (instead of 3.2 without *Rango*). For system administrators, the improvement is smaller (2.8 without *Rango*, 3.4 with *Rango*). One explanation for this observation is that system administrators lack experience with rule-based systems and, hence, find it more difficult to write such rules. Still, it can be observed that *Rango* improves the score by around 30%. Third, as far as understanding output is concerned, *Rango* is expected to be especially helpful for system administrators. The easiness of understanding LCS output increases from 2.2 out of 5 to 3.6. In comparison, the improvement for LCS experts is smaller (3.4 without *Rango*, 4.0 with *Rango*). The improvement might be smaller since the output was already understandable for LCS experts anyway. This would also match the experts' assessment in another question that the existing output (i.e., without *Rango*) already makes the learning process understandable. They rated the traceability/explainability of the learning process 4.0 out of 5 on a scale from 1 ("not at all") to 5 ("very detailed").

In addition, the LCS experts rated *Rango*'s syntax 4.2 out of 5 on a 5-point Likert scale. In the final question whether the experts would use *Rango* if available, *Rango* scored 4.2 out of 5 (scale: 1 ("no, never") to 5 ("yes, regularly")). The experts were additionally asked to suggest further features. The majority of these suggestions are already included in *Rango*'s current version but were not explicitly presented to the LCS experts in the textual description. One expert suggested that *Rango* should allow to set bounds for rule mutations. The restriction, influence, or guidance of the learning process in an LCS is an valuable addition and subject of future versions of *Rango*.

Conclusion LCS experts perceive *Rango* as useful. The above insights from Table 7.1 show that *Rango* facilitates both writing rules and understanding LCS output.

7.1.2. Usability Study

After showing that *Rango* is useful, its usability, i.e., how easily users understand and create rules in *Rango* is evaluated in the following.

Procedure To evaluate *Rango*'s usability, again an online questionnaire[1], which consists of three parts, was designed. First, participants were confronted with a textual description of a mixed-critical CPS/CPN scenario from the health care domain and the corresponding code in *Rango* that would achieve the desired adaptive behavior in the scenario. The task of the participants was to understand the code step-by-step and to answer two types of questions: (i) multiple choice questions and (ii) open questions. As far as multiple choice questions are concerned, the participants had, for instance, to choose the correct interpretation of a code snippet in natural language among four alternatives. In the open questions, participants were asked to describe the purpose of a code snippet in their own words. The participants did not receive a briefing about

[1]The whole questionnaire is available at https://forms.gle/649bMCth1F78DuPi6.

Rango's syntax or the idea behind the language during this part of the study. Analyzing the participant's performance in the first part of the study allows to answer how easy it is for potential users to write rules in *Rango*.

Second, participants were confronted with the automotive scenario that is also used as a running example to introduce *Rango* in Section 5.2.2 The task of the participants was to write the rule and its required condition, action, and sets from Section 5.2.2 step-by-step by themselves. The textual guidance led the participants through the coding process (e.g., by stating that they have to define an according set now). It additionally contained explanations of *Rango*'s syntax in a format similar to typical tutorials for programming languages available on the internet. As an example, the syntax hint for the definition of a condition in Figure 7.1 was shown. The performance of the participants in this second part of the study was analyzed to answer how easy it is for potential users to write rules in *Rango*.

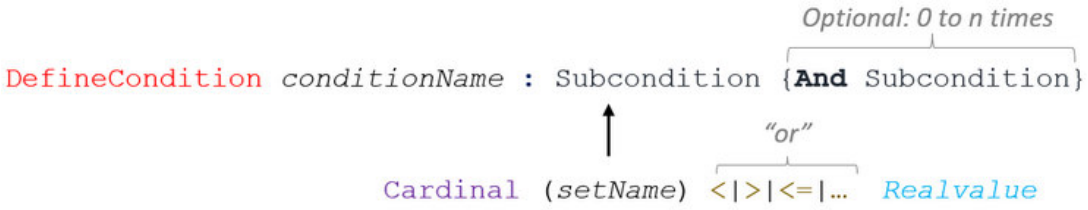


Figure 7.1: Syntax hint for the condition definition in the writing part of the usability study questionnaire.

Third, participants completed a self-evaluation related to their skills in three areas: (i) programming, (ii) computer science research related to this thesis, and (iii) enterprise information systems. The participants had to rate their expertise with regards to several programming languages, technologies, or concepts such as "Python", "Cyber-physical systems", or "ERP" on a scale from 0 ("never heard of it") to 5 ("expert").

Methodology The participants' answers to the multiple choice questions were labeled with either "correct" or "incorrect". The answers to the open questions with regards to understanding rules were categorized into "correct", "inaccurate", and "incorrect". As far as the second part of the study (rule writing) is concerned, the code that was provided for each question is labeled as either "correct", "with minor syntactic mistake(s)", "with semantic mistake(s)", or "incorrect". To analyze the third part of the study (self-evaluation), the scores that participants entered for the items in one area were averaged to obtain three values between 0 and 5 that describe the proficiency of a participant in each of the three areas (programming, research, and enterprise systems).

Participants The participants were acquired using snowball sampling. In total 37 participants (28 male, 8 female, 1 diverse) took part in the study. Their age ranges between 20 and 63 (average = 29) years. The participants had to give their highest

academic degree: 48.6% have a Master's degree, 29.7% a Bachelor's degree, 13.5% a PhD, and 8.1% no university degree. Most of the participants (62.2%) are studying or working in the field of computer science. The participants' programming experience ranges between 0 and 35 (average = 7) years. In the self-evaluation, the participants achieved an average score of 2.5 out of 5 in programming, 2.4 in computer science related to this thesis, and 1.9 in the enterprise information systems area.

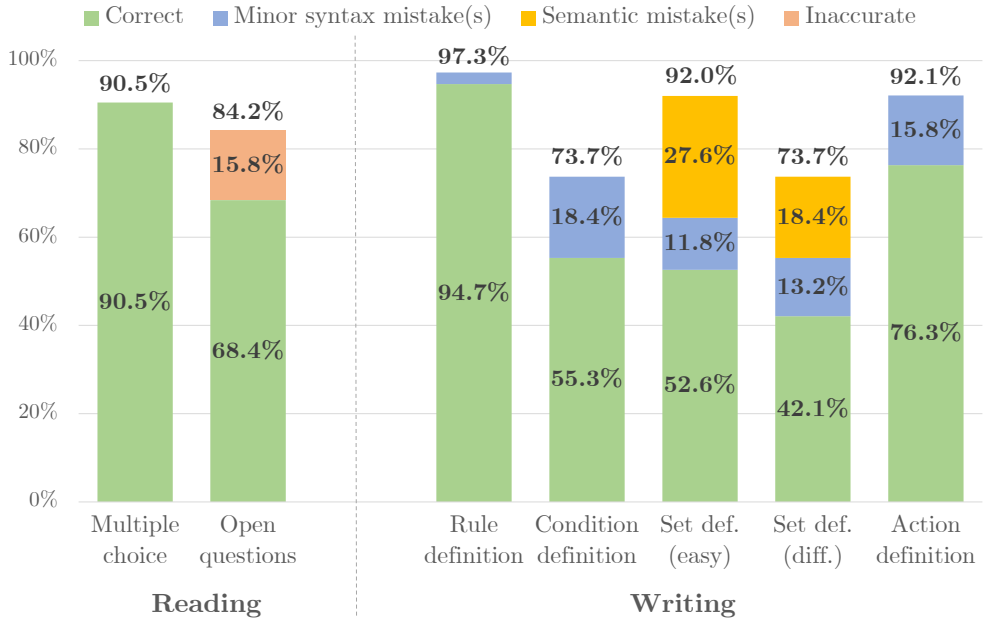


Figure 7.2: Results of the rule reading and writing part of the usability study.

Results Figure 7.2 visualizes the results of the usability evaluation. Understanding the code step-by-step and answering multiple choice questions and open question achieved a high percentage of correctness. This leads to the conclusion, that rules written in *Rango* are easy to understand and that no explicit syntax briefing is required.

In the writing part, the participants were asked to write a rule and its required condition, action and sets. In each part of the task a small number of minor syntax mistake(s) like wrong capitalization and forgotten brackets were made. In practice, the parser would note these errors and the user would get a notification to correct the syntactic mistakes. In the following, these answers are counted as correct. Furthermore, only in the definition of sets semantic mistakes were made. One common mistake was the usage of the wrong scope for the set specification. Also, in the definition of the difficult `nonLocalSuitableNodes` set, the specification of the application set was forgotten after the `Demand` attribute. Furthermore, the figure shows that the definition of the condition and sets is more demanding for the users. Still, more than half of the participants (55.3%) correctly defined the difficult set. The easier sets were correctly defined by 64.4% and the condition was formulated correctly by 73.7% of

the participants. The definition of the action was correct by 92.1% and nearly all participant defined the rule correctly (97.3%). The analysis of the data shows that even non-experts are easily able to create rules in *Rango*. Further it is useful for the application of *Rango* to explicitly explain the predefined scopes and the application of the predefined attributes in more detail to ease the definition of the conditions and sets. This was not the case in the study.

Conclusion The evaluation results above show that *Rango* is usable. Rules written in *Rango* are easy to understand even without an explicit syntax briefing. Also, even non-experts are easily able to create rule in *Rango*.

7.1.3. Performance Evaluation

So far, it has been shown that (i) experts perceive the language as a useful addition and that (ii) *Rango* is usable for potential users, even for non-experts. Now *Rango*'s performance is evaluated. Using *Rango* instead of bit strings to formulate rules introduces an overhead in terms of memory usage and computational complexity. Thus, memory usage, computational complexity of the parsing and the rule processing at runtime is examined both theoretically and in real-world measurements.

Memory Usage The rule set is parsed into an internal data structure that stores four tables: (i) rule table, (ii) condition table, (iii) action table, and (iv) set table. Those tables are linked to each other. The rule table links to the condition and action table. Those in turn link to the set table. If an action, condition, or set is referenced by name, the respective element is stored only once in the table and can be referenced from elements in the other tables. Thus, the memory requirement M is linear to the number of rules n_{rules}, conditions $n_{conditions}$, actions $n_{actions}$, and sets n_{sets} of the rule set. The memory required for a condition is linear to the number of its sub-conditions ($\mathcal{O}(n_{condition_{maxSub}})$). Similarly, the memory required for a set is linear to the number of query clauses ($\mathcal{O}(n_{set_{maxClauses}})$) describing the set. The overall memory usage M can therefore be described as:

$$M = \mathcal{O}(n_{rules} + n_{conditions} \cdot n_{condition_{maxSub}} + n_{actions} + n_{sets} \cdot n_{set_{maxClauses}}) \quad (7.1)$$

In practice, an exemplary rule set with 10 rules and respective 6 conditions with a maximum of 2 sub-conditions and 5 actions using 7 sets with a maximum of 2 query elements needs 1083 bytes in a 32 bit system and 601 bytes in a 16 bit system. This shows moderate memory needs even suitable for microcontrollers.

Parsing complexity The *Rango* parser transfers the rule set written by the system administrator into the aforementioned internal data structure. This leads to a certain delay on system startup. After that, the system works on the binary data structure

only. As the syntactic definition of *Rango* bases on a context-free LL(1)-grammar[1], the parsing complexity is $\mathcal{O}(n)$ where n denotes the number of parsed symbols [Aho+06].

Additionally, the parsing complexity is measured in practice on an Intel 11th Gen Core i7-11700 with 2.5 GHz and 64 GB RAM for rule files of different sizes. Rule sets consisting of 10, 20, 50 and 100 rules were parsed several times. The retrieved averaged parsing times for the different rule set sizes are shown in Figure 7.3.

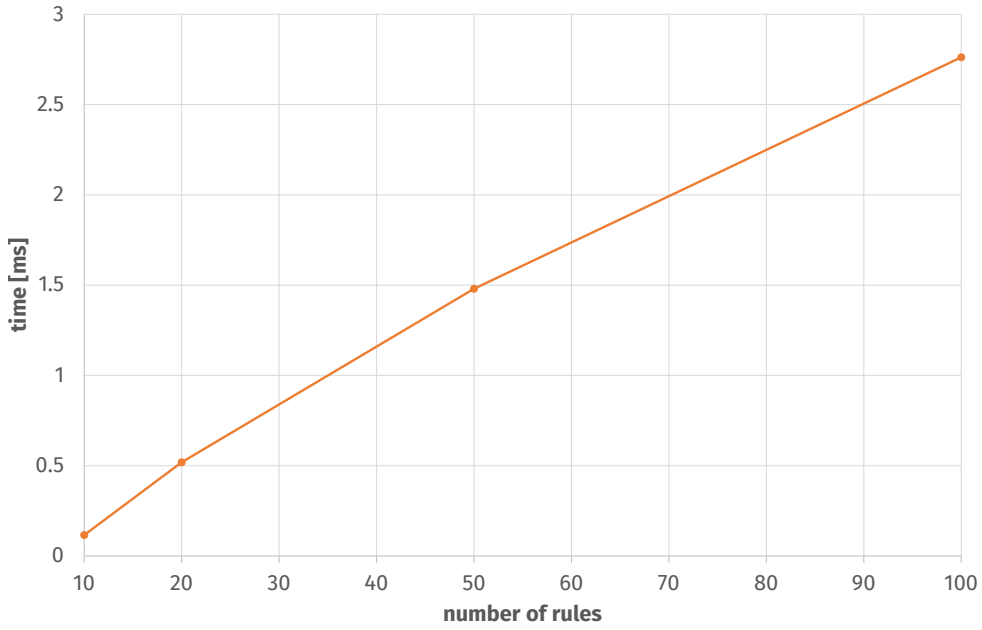


Figure 7.3: Average parsing times for rule sets of different sizes.

As expected, the parsing complexity increases approximately linear with the rule set input size and even the largest files that contain around 100 rules lead to average parsing times of less than 3 ms. On low-performance systems, an external parser on a more powerful machine could transfer the rule set into the binary representation if parsing becomes too time-consuming. This representation can then be loaded to the target system before system start.

Processing complexity At runtime, rules have to be processed, i.e., it has to be checked whether they apply to the current context. As already mentioned in Section 5.2.2, rules can be evaluated either time or event-driven. In the event-driven evaluation, the rule evaluation takes place exactly when an attribute value of a component (i.e. health, demand, ...) changes. In the case of time-driven evaluation, the rule set is evaluated periodically at fixed time intervals. Anyway, the evaluation of each rule follows the following steps: (i) determining the condition and action that belong to the rule, (ii) checking whether the condition is fulfilled, (iii) calculating each required set,

[1]In fact, it is even a LF(1) grammar. However, since the distinction between LF(1) and LL(1) is often neglected because both classes are identical, we refer to the grammar as LL(1).

or instead of calculating sets, (iv) check whether a component is element of required set(s).

In the following, the worst-case complexity ($C_{(i)}$, $C_{(ii)}$, $C_{(iii)}$) of those steps is determined . Afterwards, these are used to determine the complexity of the rule evaluation with the different strategies($C_{time-driven}$, $C_{event-driven}$).

Since the condition and action can be referenced directly from a rule in the rule table, the complexity of $C_{(i)}$ is within $\mathcal{O}(1)$. To check whether a condition is fulfilled, the given set property (cardinality, average, minimum, or maximum element) must be checked for each set of the condition. With the maximum number of sets $n_{condition_{maxSets}}$ in a condition and the maximum number of members $n_{set_{maxMember}}$ in a set, the worst-case complexity of $C_{(ii)}$ is:

$$C_{(ii)} = \mathcal{O}(n_{condition_{maxSets}} \cdot n_{set_{maxMember}}) \tag{7.2}$$

To calculate a given set, it has to be determined for each component (applications, nodes, communication channels) if is is a member of the set. *Rango* offers two options for the definition of sets. First, sets can be defined by directly referencing members by name. Second, sets can be defined via queries consisting of several clauses describing the properties of set members. If direct referencing by name is used, no calculation is needed and the complexity for the set calculation $C_{(iii)}$ is $\mathcal{O}(1)$. Else, with $n_{components}$ as the maximum number of components in the system and $n_{sets_{maxClauses}}$ as the maximum number of query clauses of a set, the worst-case complexity for calculating a set ($C_{(iii)}$) is defined by the product of both values. Combining both cases (where $n_{sets_{maxClauses}} = 0$ inicates direct referencing) leads to:

$$C_{(iii)} = \begin{cases} \mathcal{O}(n_{components} \cdot n_{sets_{maxClauses}}) & \text{if } n_{sets_{maxClauses}} > 0 \\ \mathcal{O}(1) & \text{if } n_{sets_{maxClauses}} = 0 \end{cases} \tag{7.3}$$

Instead of calculating a given set, it also can be checked if a given component is element of each set. If direct referencing by name is used in all sets, it can be directly determined if the component is part in set(s). Else, it is necessary to determine for each set if the component satisfies all queries clauses of the regarding set. Thus, with n_{sets} as the maximum number of sets in the rule set and $n_{set_{maxClauses}}$ as the maximum number of query clauses of a set the worst-case complexity to check if a component is part of set(s) ($C_{(iv)}$) is:

$$C_{(iv)} = \begin{cases} \mathcal{O}(n_{sets} \cdot n_{sets_{maxClauses}}) & \text{if } n_{sets_{maxClauses}} > 0 \\ \mathcal{O}(n_{sets}) & \text{if } n_{sets_{maxClauses}} = 0 \end{cases} \tag{7.4}$$

Based on these steps the overall complexity of event and time driven evaluation can be determined. In the time driven approach two ways on how to evaluate the rules could be distinguished. The evaluation could either calculate required sets using step (iii) n_{sets} times or instead check whether components are element of required set using step (iv) $n_{components}$ times. Both leads to the same overall complexity:

$$C_{time-driven} = \mathcal{O}(n_{rules} \cdot C_{(i)} + n_{conditions} \cdot C_{(ii)} + \underbrace{n_{sets} \cdot n_{components} \cdot n_{set_{maxClauses}}}_{n_{sets} \cdot C_{(iii)} \text{ resp. } n_{components} \cdot C_{(iv)}}) \tag{7.5}$$

An event driven evaluation takes place when an attribute of a component changes. Therefore step (iv) is used for the evaluation. The resulting complexity calculates to:

$$C_{event-driven} = \mathcal{O}(n_{rules} \cdot C_{(i)} + n_{conditions} \cdot C_{(ii)} + C_{(iv)}) \tag{7.6}$$

For the time-driven rule processing, the computational complexity occurs with a given period $t_{evaluationPeriod}$. In the event-driven processing, the frequency of attribute value changes of each component determines the computational complexity. Since *Rango* supports a large number of different attributes (cf. Section 5.2.2), together with a large number of components a high event rate can be expected. The time-driven approach instead has a well defined and controllable time behavior given by period $t_{evaluationPeriod}$ and therefore also produces less interferences (cf. Section 5.2.3). Thus, the time driven approach has been selected in this thesis[1].

If we assume a fixed size rule set (n_{rules}, $n_{conditions}$, n_{sets} is constant), the evaluation complexity scales linearly with the number of components in the CPS/CPN ($C_{time-driven} \overset{7.5}{=} \mathcal{O}(n_{components} + n_{sets_{maxMember}})$) in case of the time-driven strategies. If we instead assume a fixed environment ($n_{components}$ is constant) and a variable size rule set where the maximum number of clauses in a set and maximum number of sets in a condition is limited, the evaluation complexity scales linear with the number of rules, conditions and sets ($C_{time-driven} \overset{7.5}{=} \mathcal{O}(n_{rules} + n_{conditions} + n_{sets})$). This fact is also demonstrated by measuring the rule processing time on the aforementioned evaluation PC. The practical time required for the processing of different rule sets[2] in a fixed environment was retrieved. The resulting average times are shown in Figure 7.4. As expected, the processing time increases linear with the number of rules. Furthermore, the average processing time for rule sets of around 100 rules was well below 0.1 ms. Thus, the evaluation of *Rango* rules leads to a low overhead also tolerable on slower systems.

The above analysis furthermore shows that *Rango* on one side can handle powerful

[1] The implemented time-driven rule evaluation operates as follows: In each evaluation period all rules are processed. For each rule, the condition is checked. Once the condition is checked, the result is stored so if the same condition occurs in another rule, the condition does not have to be checked again. For each condition, the referenced sets are determined. Once a set is determined, the result is stored so if the same set occurs in another condition it does not have to be determined again. This guarantees that only the minimum necessary number of sets and conditions are processed during rule evaluation. However, in worst case, all sets and conditions must be processed once leading to the presented processing complexity.

[2] Rule sets that were consisted of 10, 20, 50 and 100 rules were evaluated multiple times resulting in about approximately 400 measurements for each rule set for the rule evaluation complexity.

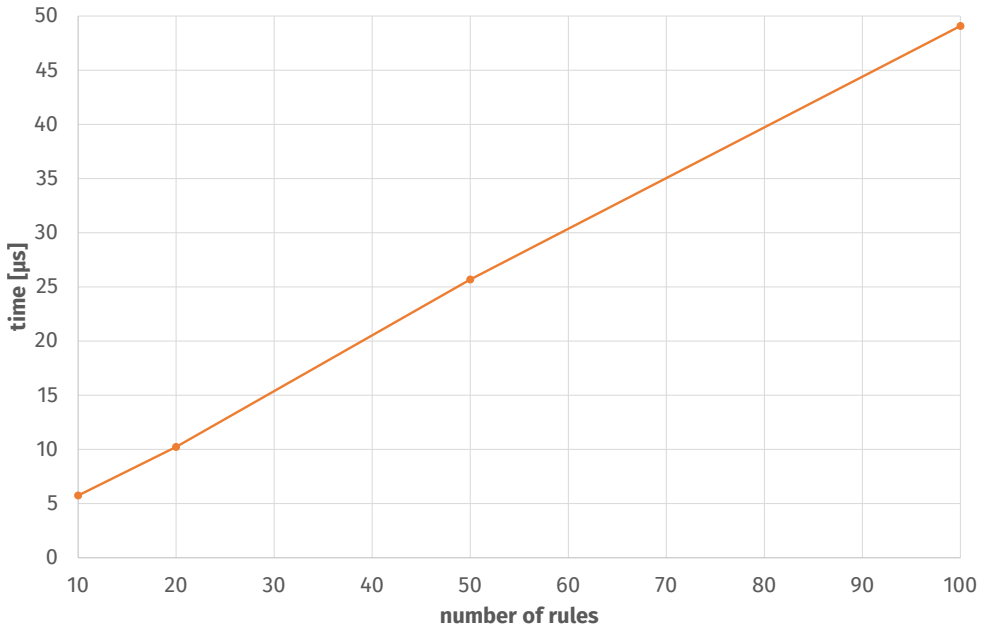


Figure 7.4: Average rule evaluation complexity times for rule sets of different sizes.

and flexible set-based rules with appropriate processing complexity, while on the other side also is able to efficiently process simple rules using direct references. According to equation 7.5, *Rango* is able to process a rule set consisting of such simple rules ($n_{rules} = n_{conditions} = n_{sets}$; $n_{condition_{maxSets}} = n_{set_{maxMember}} = 1$; $n_{sets_{maxClauses}} = 0$)[1] in $\mathcal{O}(n_{rules})$. This is the same complexity a limited language providing such simple rules only would have. Thus, *Rango* offers the advantage of flexibility without resigning efficiency.

7.2. Real-Time Behavior of *Chameleon*

The real-time behavior of *Chameleon* divides into two parts: (i) the real-time behavior of the adaptation mechanism and (ii) the real-time behavior of the application execution.

7.2.1. Real-Time Behavior of the Adaptation Mechanism

The rule engine defines the real-time behavior of the adaptation mechanism. It processes all rules periodically[2] on each instance of the *Chameleon* middleware with the evaluation period $t_{\texttt{EvaluationPeriod}}$ (cf. Section 5.2.2). Once a rule is executed, the reward is calculated after the reward delay. During that time, further rule processing

[1] Each rule in the system is associated with a single condition that contains a unique set, and within that set, there is precisely one directly referenced component.

[2] Compares all rule conditions to the current system/component condition to determine if adaptation is necessary.

is delayed. The reward delay has a global default value $t_{\texttt{RewardDelay}}$ (cf. Section 5.2.2), but can also be individually defined for each rule r to $t_{\texttt{RewardDelay}_r}$. Furthermore, the monitoring period of components (applications, nodes, communication channels) have to be taken into account to detect changed system states. For remote components, also the communication time influences the real-time behavior.

In the following, the reaction time the rule engine needs from an event triggering system adaptation to the execution of the first adaptation rule is determined. Afterwards, the occurrence of critical races, how to solve this issue and the influence of the solution on the reaction time is introduced. Finally, the resulting adaptation time is analyzed.

Reaction Time First, the reaction time the rule engine needs from an event triggering system adaptation to the execution of the first adaptation rule is determined. Let us initially assume there is no reward delay pending. This is true if the system has no adaptation action yet or at least for $\max_{\forall r}(t_{\texttt{RewardDelay}}, t_{\texttt{RewardDelay}_r})$. It is necessary to distinguish between (i) local and (ii) remote components. For local components, the local monitoring periods and the evaluation period influence the reaction time. An event changing the state of a component can occur asynchronously at any time within the local monitoring period. Therefore, if it takes the full duration of the monitoring period to detect the change, in worst case the change is detected at the end of the next monitoring period. This change then is registered by the rule engine at the beginning of the next evaluation period. Therefore, if a state change of a local component j triggers an adaptation (e.g. a local application j gets unhealthy), the worst case reaction time $t_{ReactionTime_{local_j}}$ calculates to:

$$t_{ReactionTime_{local_j}} \leq t_{\texttt{EvaluationPeriod}} + 2 \cdot t_{MonitoringPeriod_{local_j}} \tag{7.7}$$

For some events, the full duration of $n_{monitorPeriods}$ (instead of one as above) monitoring periods is necessary to detect a state change[1]. So the above equation can be generalized:

$$\begin{aligned} t_{ReactionTime_{local_j}} \leq & t_{\texttt{EvaluationPeriod}} + \\ & (n_{monitorPeriods} + 1) \cdot t_{MonitoringPeriod_{local_j}} \end{aligned} \tag{7.8}$$

In case of a remote component, it takes additional time to transport the monitored data to the destination. This time consists of two parts, the remote monitoring period $t_{MonitoringPeriod_{remote}}$ (since local monitoring might have detected a state change immediately after the last remote monitoring period which is a multiple of the local monitoring period) and the communication time $t_{Communication}$. Thus, if the state change of a remote component k triggers an adaptation, (e.g. a remote node k gets

[1] This is e.g. true for events raising health values. To avoid trashing, the worst of the last $n_{monitorPeriods} = 5$ monitored health values is used in the current implementation of the Analyze stage of the MAPE-K loop. This enables an immediate detection of a dropping health value while raising health values need to be confirmed during 5 monitoring cycles.

unhealthy), the reaction time $t_{ReactionTime_{remote_k}}$ calculates to:

$$t_{ReactionTime_{remote_k}} \leq t_{MonitoringPeriod_{remote_k}} + t_{Communication_k} + t_{ReactionTime_{local_k}} \tag{7.9}$$

After n local monitoring cycles, the monitored data is sent to remote parts of the system. Therefore, the remote monitoring period of a component is n times its local monitoring period[1]. Using this, the reaction time for remote components can be expressed by:

$$\begin{aligned} t_{ReactionTime_{remote_k}} &\leq n \cdot t_{MonitoringPeriod_{local_k}} + t_{Communication_k} + t_{ReactionTime_{local_k}} \\ &\overset{7.8}{=} (n + n_{monitorPeriods} + 1) \cdot t_{MonitoringPeriod_{local_k}} + t_{Communication_k} + t_{\texttt{EvaluationPeriod}} \end{aligned} \tag{7.10}$$

Now, if the state change of multiple local and remote components (e.g. a local application j gets unhealthy and a remote node k gets unhealthy) triggers the condition of a rule, the overall reaction time is defined by the maximum of the reaction times for the triggering components:

$$t_{ReactionTime} \leq \max_{\forall j,k \in \text{triggering components}} (t_{ReactionTime_{local_j}}, t_{ReactionTime_{remote_k}}) \tag{7.11}$$

A special case are crashed components. Here, the crash is detected by missing life signs, ie. remote monitoring data. A component is assumed to be crashed if life signs are missed for more than $l_{LifeSignLimit}$ remote monitoring periods. Therefore, the reaction time to crashes of a component k calculates to:

$$t_{ReactionTime_{Crash_k}} \leq l_{LifeSignLimit} \cdot t_{MonitoringPeriod_{remote_k}} + t_{Communication_k} + t_{\texttt{EvaluationPeriod}} \tag{7.12}$$

To derive a suitable evaluation period for the following practical experiments (cf. Section 7.3), on the one hand the local monitoring periods in the system can be a guideline. For performance reason, it is advisable to chose the evaluation period smaller or at least equal to the smallest local monitoring period in the system:

$$t_{\texttt{EvaluationPeriod}} \leq \min_{\forall i \in \text{system components}} (t_{MonitoringPeriod_{local_i}}) \tag{7.13}$$

On the other hand, the evaluation period has to be much greater than the rule processing time. The time complexity of rule processing is given in Section 7.1.3. On the target system used, this time is well below $0.1ms$ for 100 rules. The local monitoring periods can be configured and currently are chosen dependent on the application period and

[1] The size of n is the tradeoff between communication overhead and the timeliness of remote monitoring data.

node and communication channel performance.

For all scenarios used in the following further evaluation of *Chameleon*, the evaluation period $t_{\texttt{EvaluationPeriod}}$ has been set to $100ms$ in the rule file configuration directives. This value is far above the rule processing time and below the minimum local monitoring period of $125ms$ in all conducted experiments. If faster reaction times are required, these periods and times can be reduced at the cost of higher computational load and communication overhead.

As an example for local and remote reaction times, let us assume that we have an application component, where the local monitoring period is configured to 5-times of its execution period. With an execution period of $25ms$ this leads to a monitoring period of $125ms$ like mentioned above. Furthermore let $n_{monitorPeriods}$ be 1 (as it is for most of the events). If the application is local, $t_{ReactionTime_{local}}$ calculates to:

$$t_{ReactionTime_{local}} \leq 100ms + 2 \cdot 125ms = 350ms \tag{7.14}$$

Else if the application is remote, where the remote monitoring period is $n = 2$ times of the local monitoring period and the communication time is $t_{Communication} = 50ms$, $t_{ReactionTime_{remote}}$ calculates to:

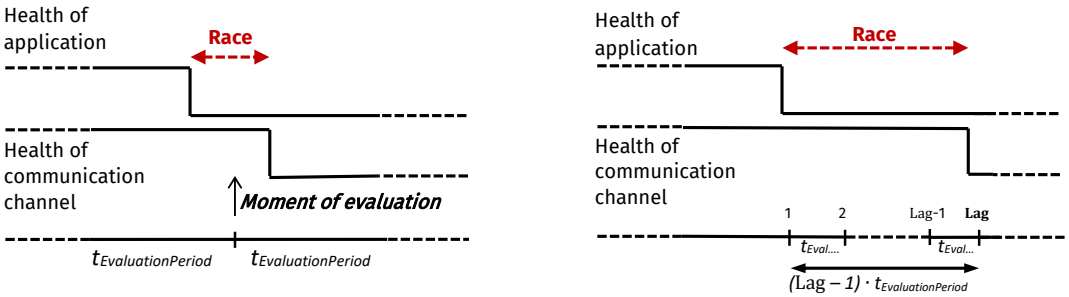
$$t_{ReactionTime_{remote}} \leq 100ms + 50ms + 4 \cdot 125ms = 650ms \tag{7.15}$$

Critical Races and `Lag` If an event causes state changes of multiple components, races can occur. Due to communication times and different monitoring periods, these changes will not propagate at once resulting in short-term intermediate state values. A race becomes critical, if rules exist which trigger on such intermediate state values. These rules then would trigger unintended. As an example, let us assume an event causes an application and a communication channel to both become unhealthy. If the unhealthy state of the application is propagated a bit quicker than the one of the communication channel, there will be the intermediate situation where the application is marked as unhealthy while the communication channel is still indicated as healthy. A rule with a condition for an unhealthy application in combination with healthy communication channel will then trigger unintended.

This issue cannot be handled by simply extending the evaluation period, since independent of its duration in worst-case the next rule evaluation can take place during a critical race, see Figure 7.5a. However, repeated condition checking can solve this issue, as shown in Figure 7.5b. Thus, *Rango* introduces the option to define a `Lag` for rules (cf. Appendix A). The `Lag` is the number of evaluation periods a rule condition has to be repeatedly true before the condition triggers. Then a critical race with a duration of t_{Race} does not unintendedly trigger a rule if the following holds true:

$$t_{Race} \leq (\texttt{Lag} - 1) \cdot t_{\texttt{EvaluationPeriod}} \tag{7.16}$$

The reaction time for a rule i with `Lag` then increases from the reaction time without



(a) Rule evaluation during race.

(b) Covering a race by repeated rule evaluation

Figure 7.5: Critical races and `Lag`.

`Lag` (`Lag` $= 1$) to:

$$t_{ReactionTime_i} = t_{ReactionTime_{without\texttt{Lag}}} + (\texttt{Lag}_i - 1) \cdot t_{\texttt{EvaluationPeriod}} \tag{7.17}$$

Since such races are caused by different communication times and monitoring periods, an upper bound for t_{Race} can be given for a rule affected by a critical race. Assuming c is one of the components which trigger a rule based on a critical race, the propagation time for the state of this component calculates to:

$$t_{Propagation_c} = \begin{cases} t_{MonitoringPeriod_{local_c}} & \text{if } c \text{ is local} \\ t_{MonitoringPeriod_{remote_c}} + t_{Communication_c} & \text{if } c \text{ is remote} \end{cases} \tag{7.18}$$

The maximum possible race duration is then given by the biggest propagation time difference $t_{Propagation_{diff}}$ of the components triggering the rule:

$$\begin{aligned} t_{Race} \leq t_{Propagation_{diff}} = &\max_{\forall c \in \text{triggering components}} (t_{Propagation_c}) - \\ &\min_{\forall c \in \text{triggering components}} (t_{Propagation_c}) \end{aligned} \tag{7.19}$$

Using the two Equations 7.16 and 7.19 for t_{Race}, a lower limit for `Lag` in rules with critical races can be given:

$$\texttt{Lag} \geq \left\lceil \frac{t_{Propagation_{diff}}}{t_{\texttt{EvaluationPeriod}}} \right\rceil + 1 \tag{7.20}$$

Let us retrieve the above example of a rule, which checks for an unhealthy application in combination with a healthy communication channel (cf. Appendix B.5). Assuming the application and the communication channel are local and the local monitoring period of the application is $100ms$ while the monitoring period of the communication channel is $200ms$, the maximum possible duration of the race calculates to $200ms - 100ms =$

$100ms$. With an evaluation period of $100ms$, the necessary `Lag` for this rule according to Equation 7.20 is 2.

Adaptation Time Based on the reaction time and the reward delay, the adaptation time can be determined. Let us assume the triggering state leads to an adaptation sequence of x rule executions. Let us further assume no other event triggering an adaptation occurs during that sequence. Then, the first rule is executed after the reaction time while the remaining $x-1$ rules are executed after the reward delay and an optional lag for each rule. The adaptation time therefore calculates to:

$$t_{Adaptation} \leq t_{ReactionTime_1} + \sum_{i=1}^{x-1} (t_{\texttt{RewardDelay}_i} + (\texttt{Lag}_{i+1} - 1) \cdot t_{\texttt{EvaluationPeriod}}) \tag{7.21}$$

If the rules executed have no individual reward delay and no lag ($\texttt{Lag} = 1$), this formula can be simplified to:

$$t_{Adaptation} \leq t_{ReactionTime_1} + (x-1) \cdot t_{\texttt{RewardDelay}} \tag{7.22}$$

Now, the initial assumption is relaxed that no reward delay is pending when an adaptation is triggered. Furthermore, another triggering event is allowed to occur during the execution of an adaptation sequence. Both relaxations lead to a possible combination of adaptation sequences. If such a combination arises, it can be analyzed as a single sequence consisting of the combined adaptation actions. Therefore, if s sequences are combined, then the worst-case time can be calculated by setting $x = \sum_{i=1}^{s} x_i$ in the above formulas.

The reward delay depends on the time necessary to observe the outcome of an action. This is influenced by the kind of the action, how many local monitoring periods it takes to observe its outcome, the remote monitoring period to publish this system wide as well as the communication time in case of a global action. Let us assume the action of rule r modifies the behavior of component g. Let $n_{monitorPeriods}$ be again the number of local monitoring periods it takes to observe its outcome and the remote monitoring period again be n times the local monitoring period. Then the reward delay for an action a can be calculated in similar way as the reaction time to:

$$\begin{aligned} t_{\texttt{RewardDelay}_a} =& (n + n_{monitorPeriods} + 1) \cdot t_{MonitoringPeriod_{local_g}} + \\ & t_{Communication_g} + t_{\texttt{EvaluationPeriod}} \end{aligned} \tag{7.23}$$

The global reward delay then can be set to:

$$t_{\texttt{RewardDelay}} \geq \max_{\forall a \in \text{rules}} (t_{\texttt{RewardDelay}_i}) \tag{7.24}$$

Taking for example a local monitoring period of $125ms$, a communication time

of $50ms$, n being again 2 and setting $n_{monitorPeriods}$ to the large value of 5[1], this would result in a reward delay of $1150ms$. A local monitoring period of $200ms$ with the same other parameters would result in $1750ms$ reward delay. Aligned to these value, the global reward delay `RewardDelay` has been set to $2000ms$ in the rule files configuration directives for the following experiments. For specific actions like e.g. pausing unimportant applications and restarting dead applications, a shorter individual reward delay ($300ms$, see also Appendix B.5) can be chosen, because $n_{monitorPeriods}$ is much smaller (mostly 1). This e.g. allows in case of an overload situation to pause an unimportant application (to make room) and to restart an important crashed application much quicker as with the global reward delay.

7.2.2. Real-Time Behavior of the Application Execution

The real-time properties of application execution can be monitored by the rule engine and influenced by actions. Based on observable timing parameters like the rate of missed timing constraints, the distances to timing constraints, the allowed missrates and allowed distances (cf. Section 5.1), the related health values (cf Section 5.2.1), and with respect of the application importance the system load can be modified. Possible actions are the change of priorities, periods, deadlines, scheduling strategies as well as the relocation and pausing of applications (cf. Sections 4.4 and 5.4). A major goal of the adaptation mechanism is to keep the timing constraints of all applications. If this is not possible due to an overload situation, it aims to keep the constraints of the applications ordered by their importance. Besides the observable parameters and applicable actions, the applications are black boxes to the middleware. The overall behavior of the adaptation mechanism is evaluated in the next section.

7.3. Evaluation of the *Chameleon* Rule Based MAPE-K Adaptation Mechanism

The usability of the rule-based MAPE-K adaptation mechanism for managing MC-CPN's and their complexity while considering the mixed-criticality is investigated in an automotive application scenario which is introduced in Section 7.3.1. Afterwards, a basic rule set is presented which is used for almost all evaluation scenarios. Finally, nine evaluations are conducted. Each evaluation has been repeated multiple times with completely reproducible results.

7.3.1. Application Scenario and Configuration

An application scenario suitable to evaluate the adaptation mechanism of *Chameleon* and *Rango* has to be a CPN with mixed critical timing constraints and dynamic behavior. The field of automotive applications is an example which fulfills these requirements. A car can be seen as a CPS containing embedded components with a wide range of

[1] As applied for raising health values, see before.

criticality and timing constraints (from low-critical comfort functions like infotainment via mid-critical functions like driving assistants to highly safety critical components like *Brakes* or *Steering*). Dynamic behavior results from activating or deactivating such components at run time (turning on or off e.g. *Infotainmant*, *Navigation* or a driving assistant like *Cruise Control*) or changes in the hardware environment caused e.g. by processor problems or failures. Cooperating cars (e.g. in platooning) can bee then seen as a CPN. This introduces more dynamics by cars entering or leaving a platoon and at the same time offers more opportunities for adaptation since functionalities can be shared or transferred between cars. Therefore, such an automotive scenario has been chosen for the evaluations.

In classic car configurations, each functionality has its on processor (ECU). However, this produces the most hardware costs, especially when it comes to redundancy. To prevent a functionality from failing, the corresponding ECU has to be doubled by a backup ECU. Therefore, more recent car configurations use another approach. Here, different functionalities share more powerful processors. Each of these processors host more than one functionality. This reduces the costs and eases redundancy, since functionalities now can be moved between processors. It also offers more opportunities for adaptation.

Still, in today's configurations highly safety critical functions are usually kept separate from low and medium critical functions. For the evaluations presented here, even this restriction is relaxed. All functions can share the entire processor set. This has the following reasons: First, it allows for the assessment of the extent to which the adaptation mechanism of *Chameleon* and *Rango* can effectively handle mixed critical applications with a wide range of criticality levels. Second, it gives the maximum adaptation space and therefore offers the biggest potential to keep the system healthy. In a simulation environment, this can be easily explored since deficiencies will not cause any physical harm. In a real environment, this approach could also be used if the highly safety critical functions are backed by an emergency mode (e.g. a simple low level direct brake mechanism is available as long as the high level anti locking brake function is unavailable).

As basis, a vehicle with the common functions available in a today's cars has been chosen. Figure 7.6 shows its architecture in the frame of the system model. Thereby, all gray marked applications are started on driver demand (dynamics), all other applications are activated at system start. Furthermore, it is designed so that each processor has access to each other and each sensor and actuator thus allowing the applications to share the entire processor set. For the evaluation of more than one vehicle, $NetworkNode_D$ additionally offers a wireless communication link ($Comm_2$) to other vehicles.

Application Layer On the application layer, several common applications available in today's cars are present. Table 7.2 outlines the main parameters of these applications. It shows the importance of an application as well as its period and priority, which has been chosen according to rate monotonic scheduling (RMS, $priority \widehat{=} 1/period$).

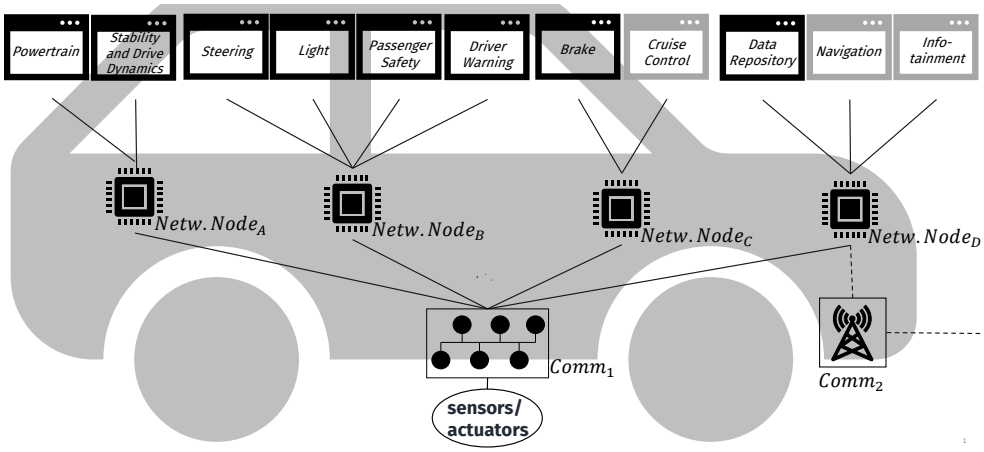


Figure 7.6: Evaluation Scenario. Gray marked applications are started on driver demand (dynamics), all other applications are activated at system start.

Furthermore, it gives the maximum allowed period for tuning (so the quality of less important applications can be reduced to reduce their load for the system, - means no period tuning allowed), the maximum allowed data compression rate (- means no data compression) and the computational and data load range. The XML code of the applications used in the simulator can be found in Appendix D.

The local monitoring period has been configured to 5-times the application period, while the global monitoring period is 2-times the local monitoring period. These values are a suitable compromise between monitoring resolution (as lower the values as better) and overhead (as higher the values as better).

In the following, the various interactions[1] of the applications with each other as well as with sensors and actuators is briefly described:

- ***Steering*** is one of the three most important applications. It is responsible for steering support of the driver (power steering, steering operations to ensure stability, ...). It gets data from the steering sensors (steering wheel sensors, steering angle sensors, ..) and delivers data to the steering actuators (steering wheel motors). Additionally, the application *Stability and Drive Dynamics* interacts with the application *Steering* by remote method invocation. *Steering* is activated at system start.

- ***Brake*** is another one of the most important applications. It is responsible for brake support of the driver (anti-locking brake, brake operations to ensure stability, ...). It gets data from the brake sensors (wheel sensors, brake pedal sensors, ...), delivers data to the brake actuators (brake calipers, cylinders, ...) and messages to the application *Lights* to trigger the brake light. Additionally,

[1]Interactions which wait for return values all have a timeout. So in case the called application is not available, the calling application can run with reduced functionality.

the application *Stability and Drive Dynamics* interacts with the application *Brake* by remote method invocation. *Brake* is activated at system start.

- ***Passenger Safety*** is the third of the most important applications. It handles safety functions like airbags or seatbelt pretensioners. It gets data from the safety sensors (e.g. crash sensors) and delivers data to the safety actuators (e.g. airbag ingnition). Additionally, it handles messages from the applications *Stability and Drive Dynamic*s and *Driver Warning* to rate the current driving situation. Finally, it exchanges data via the central application *Data Repository*. *Passenger Safety* is activated at system start.

- ***Powertrain*** is responsible for the engine control. Therefore, it is also a very important application, but is set here one level below the previous ones, since an engine failure is usually a bit less critical than a *Brake* or *Steering* failure. It gets data from various engine sensors (e.g. air intake and pressure, crankshaft position, ...) and delivers data to engine actuators (e.g. fuel injection, ignition, ...). Additionally, the application *Stability and Drive Dynamics* interacts with the application *Powertrain* by remote method invocation, e.g to reduce the engine power in critical situations. *Powertrain* is activated at system start.

- ***Lights*** is an application set on the same importance level as *Powertrain* in this scenario. It is responsible for lighting the vehicle (main lights, brake lights, direction indicators, ...). It gets data from the light sensors (e.g. lightness) and delivers data to the light actuators (the lights themselves). It receives messages from the application *Brake* to handle the brake lights and exchanges data via the central application *Data Repository*. *Lights* is activated at system start.

- ***Stability and Drive Dynamics*** is responsible for handling electronic stability and dynamics functions like dynamic traction control or dynamic stability control. In this scenario, the importance is set one level below *PowerTrain* and *Lights*, since a careful driver is able to drive safely also without these features. *Stabilty and Drive Dynamics* interacts with the applications *Steering*, *Brake* and *Powertrain* by remote method invocation to intercept in case of a critical situation. The applications *Cruise Control* and *Driver Warning* also interact with *Stability and Drive Dynamics* by remote method invocation to execute driving commands. Finally, it sends messages to the application *Passenger Safety* to inform about the current driving situation and exchanges data via *Data Repository*. *Stability and Drive Dynamics* is activated at system start.

- ***Data Repository*** is the central service for data exchange. It is the only reactive application (all the others are periodic) in this scenario and its importance level has been chosen the same as *Stability and Drive Dynamics*. Many of the other applications use this service. *Data Repository* is activated at system start.

- ***Cruise Control*** is an application realizing functions like automatic distance keeping. It receives data from obstacle sensors (e.g. Lidar or Radar) and controls

the speed via remote method invocation of the application *Stability and Drive Dynamics*. Its importance has been chosen one level below *Stabilty and Drive Dynamics*, since it is more comfort oriented. Also, *Cruise Control* exchanges data via *Data Repository*. *Cruise Control* is activated on driver demand.

- ***Driver Warning*** informs the driver if he is not keeping the right distance to the vehicle ahead. This is also regarded here as a comfort function, as a careful driver should be able to manage this on his own. Therefore, its importance is set below *Cruise Control*. It uses the same obstacle sensors as *Cruise Control* and interacts with *Passenger Safety*, *Stability and Drive Dynamics* and *Data Repository*. *Driver Warning* is activated at system start.
- ***Navigation*** calculates routes and guides the driver to his target. This is mainly a comfort function, therefore the importance has been chosen below *Driver Warning*. It receives data from the GPS sensors and interacts with the driver via the display and speaker actuators. Furthermore, it exchanges data with other applications via the *Data Repository*. *Navigation* is activated on driver demand.
- ***Infotainment*** is a comfort application providing the driver with various information and entertainment. It has assigned the lowest importance level. It receives data from the radio antenna sensors and uses the speaker and display actuators. Furthermore, it exchanges data with other applications via the *Data Repository*. *Infotainment* is activated on driver demand.

Please note that the importance values for the applications have been reasonably chosen, but other *importance* assignments would also be possible[1]. However, for the evaluation it is only necessary that different *importance* values exist to assess the influence of mixed-criticallity to the adaptation mechanism.

[1]Remember: Importance must no be confused with priority (cf. Section 3). While *importance* refers of the level of criticality of a task in the system, priority refers to the order in which tasks are scheduled based on the period or deadline. This is also clearly shown in the Table 7.2.

Application	Importance	Priority	Period [ms]	Max Period [ms] (tune)	Instructions* Max (Min)	Data Sent** Max (Min)	Data Received** Max (Min)	Max Compression
Steering	7	4	50	50 (-)	200000 (180000)	268 (128)	368 (128)	-
Brake	7	4	50	50 (-)	200000 (180000)	640 (160)	640 (160)	-
Passenger Safety	7	5	25	25 (-)	25600 (10500)	580 (0)	1680 (240)	-
Powertrain	6	5	25	25 (-)	100000 (80000)	720 (240)	960 (480)	-
Lights	6	1	500	1000 (2)	105000 (80000)	800 (800)	832 (800)	-
Stability and Drive Dynamics	5	4	50	100 (2)	270100 (190100)	2670 (1230)	1640 (1200)	-
Data Repository	5	i ***	-	- (-)	125000 (125000)	480 (480)	480 (480)	-
Cruise Control	4	4	50	100 (2)	300000 (260000)	960 (480)	12960 (12480)	-
Driver Warning	3	2	250	500 (2)	250100 (250100)	990 (990)	12960 (12960)	-
Navigation	2	3	100	500 (5)	300000 (260000)	6480 (6000)	6480 (600)	0.8
Infotainment	1	4	50	500 (10)	84000 (44000)	6480 (6000)	6584 (6104)	0.8

Table 7.2: Application parameters. (* instructions per period (or invocation in case of the reactive Data Repository); ** bits per period (or invocation in case of the reactive Data Repository); *** inherits priority from caller)

Network Node Layer The vehicle configuration contains four networked computation nodes, each of them a single threaded processor with a computational performance of 10 million instructions per second (MIPS). This resembles the typical performance of mid-size microcontrollers which are quite common in automotive ECUs. As scheduling scheme for the applications, fixed priority preemptive (FPP) scheduling is used. This is also typical for automotive real-time ECUs [But11]. With four computation nodes, the overall processing power is 40 MIPS. From Table 7.2, the maximum processing demand of an application can be estimated by dividing the maximum number of instructions by its period. Summing up the demand of all applications[1] results in an overall maximum demand of about 31.6 MIPS. On one side, this results in an overall computational load of 79% which is a bit above the Liu/Layland bound [LL73] for a single processor ($n((2^{1/n}) - 1) = 71\%$ for $n = 11$ applications) to guarantee all timing constraints. Furthermore, due to the multi-processor configuration, not all application/node combinations will work. So when all applications are running, in worst-case the computational power might become a bottleneck. This becomes even more true in case of node failures or slow-downs (e.g. due to thermal issues). On the other side, it is a main scope of this thesis to evaluate how the proposed adaptation mechanisms of the MAPE-K loop can handle such a bottleneck. Furthermore, for economic reasons as less hardware resources as possible should be used. Therefore, a tight configuration of four computation nodes have been chosen for the evaluation. The initial assignment of applications to nodes can also be seen in Figure 7.6.

Network Layer As communication network in the vehicle, a bus ($Comm_1$ in Figure 7.6) with a data transmission capacity of 500000 bits/sec interconnects the computations nodes, sensors and actuator. The bus also enables FPP scheduling for data transmission. This represents the capabilities of a CAN bus, which is frequently used in automotive configurations. From Table 7.2, also the maximum communication demand of the interaction between applications, sensors and actuators can be estimated in the same way as the computational demand. Building the sum of sent and received data over all applications results in an overall maximum demand of about 520000 bits/sec. This is a worst case load of 104%. So even a bit more than the computation nodes, the communication bus in worst case might become a bottleneck. For the same reasons as above, this enables the evaluation of how the MAPE-K loop can handle this and saves hardware resources. For inter-car-communication, a wireless connection ($Comm_2$) with a maximum datarate of 1000000bits/s is established. This datarate is oriented on the speed of a very low performance IEEE 802.11p WLAN.

Sensor/Actuator Layer The available sensors and actuators and their purpose have already been mentioned in the interaction description of the application layer. All the sensors used are reactive sensors with priority inheritance. This means they are triggerd by the applications and deliver sensor data with the same priority as the

[1] The maximum invocation period of the reactive application *DataRepository* is assumed to $100ms$ here.

triggering application. The actuators of course receive their data also with the priority of the sending application. Table 7.3 sketches the data packet sizes of the sensors and actuators used in the simulator.

Sensors	Packet size [bits]	Actuator	Packet size [bits]
Steering	128	Steering	128
Brake	160	Brake	128
Safety	240	Safety	240
Lights	320	Lights	320
Engine	480	Engine	240
GPS	6000	Display/Speaker	6000
Radio	6104		
Obstacle	21000		

Table 7.3: Sensor/actuator packet sizes in the simulator.

7.3.2. Basic Rule Set

Rango allows to write generic as well as application specific rules. If not otherwise noted, a completely generic basic rule set is used for the following evaluations. This rule set can be found in Appendix B. Hereafter, basic principles of the rule set are explained.

The rules use the *local* condition *global* action scheme (cf. Table 5.5). Local condition means that at least one subcondition, usually the trigger, is purely local. To demonstrate this, a relocation rule from the basic rule set is used as example[1]. This rule has the following condition:

```
DefineCondition RelocationMightBeUseful :
Cardinal (LocalUnhealthyApps) > 0 And
Cardinal (LocalUnhealthyNode) > 0 And
Cardinal (NonLocalSuitableNode) > 0 And
Cardinal (LocalUnhealthyComm) = 0
Note[2] "If there are unhealthy apps on an unhealthy
node, but comm is ok and there are powerful nodes
available, a relocation might be useful"
```

The trigger part of the condition is that the number of local unhealthy applications which are not paused is above 0:

```
Cardinal (LocalUnhealthyApps) > 0
```

with:

[1] This is an extension of the relocation rule that was already used to introduce *Rango* in Section 5.2.2.

[2] `Note` is a specific form of a comment. While a comment (encapsulated by /* */) is simply ignored during parsing, a note is stored with the statement. Therefore, when the rules are written back with modified rewards after learning, notes are maintained while comments are lost.

```
DefineSet LocalUnhealthyApps :
App Local Where Health < 0, PeriodTune >= 0
Note "Local Apps with health value below 0
(and not paused)"
```

There are two more local sub-conditions that require the local node to be unhealthy while the local communication channels are healthy:

```
Cardinal (LocalUnhealthyNode) > 0 And
Cardinal (LocalUnhealthyComm) = 0
```

with:

```
DefineSet LocalUnhealthyNode :
Node Local Where Health <= 0.1
Note "Local node with health value of 0.1
and below"

DefineSet LocalUnhealthyComm :
Comm Local Where Health <= 0.15
Note "Local comm with health value of 0.15
and below"
```

The remaining sub-condition is global and requests a node suitable for the relocation to be present somewhere else in the system:

```
Cardinal (NonLocalSuitableNode) > 0
```

with:

```
DefineSet NonLocalSuitableNode :
Node NonLocal Where
Capacity % >= Demand + 0.15 (LocalUnhealthyApps),
LifeSignAge <= Limit, Health Max
Note "Non local alive node with enough capacity
(leave 15% free) to take an unhealthy app"
```

This set selects non local nodes with enough capacity to take the first unhealthy application in the set LocalUnhealthyApps. Since fixed priority preemptive scheduling cannot guarantee a 100% node load, a spare capacity of 15% is left free. Furthermore, only alive nodes are considered. If more than one of such nodes exist the one with the maximum health is taken.

The local parts of the condition assure that only the local *Chameleon* middleware can trigger this condition thus avoiding conflicts, when several *Chameleon* instances would trigger the same rule for the same components. The global parts of the condition take care for the global knowledge necessary to make good global decisions.

The action taken for this condition is global:

```
DefineAction RelocateLocalUnhealthyApp :
Relocate LocalUnhealthyApps To NonLocalSuitableNode
```

```
Note "Relocate one local unhealthy app to a
suitable node"
```

The entire relocation rule then is as follows:

```
If RelocationMightBeUseful
Then RelocateLocalUnhealthyApp
Note "Rule for relocation"
```

The rule set itself contains 17 rules with 15 actions, 17 conditions and 23 sets. It is structured into groups of rules for different purposes (see Appendix B.5).

The first group of three rules is dedicated to handle crashes of nodes and applications.

1. Restart dead applications:
   - Applications are monitored for life sign ages, and if an application exceeds its designated life sign age, it is considered dead.
   - The rule automatically restarts dead applications, ensuring continuous operation and recovery from crashes.
   - Dead applications are ordered based on their *importance*, with higher important applications being restarted first. When multiple applications crash simultaneously, the restarting process follows a strict descending order of *importance*. This ordering ensures that critical applications are recovered first.

2.&3. Make space by tuning (second rule of the group) or pausing (third rule of the group) lower important applications:
   - In cases where there is insufficient capacity to restart a higher important crashed application, tuning or pausing lower important applications is employed.
   - Tuning is preferred over pausing as long as possible, aiming to reduce the performance of lower important applications while freeing up resources.

To make space, applications are tuned or paused beginning with the lowest importance in strictly ascending order of *importance*[1][2]. Thus, this group of three rules ensures that applications are kept alive in the order of their importance at the cost of lower important applications if necessary in an overload situation. If n high important applications have to be restarted while m low important applications have to be tuned or paused for this, then it takes $n + m$ rule executions to perform this task[3]. Also, these rules have the following local sub-condition to avoid conflicts.

[1] Pausing is done regardless if an application is crashed or not, thus ensuring the applications are paused in strictly ascending order of *importance* to make room.

[2] A similar approach is used in [Hut22]. However, only pausing and no tuning is performed there.

[3] Application specific rules can further speed up the recovery of extremely high important applications (cf. Section 7.3.8).

```
Cardinal (AliveNodeWithLowestIdIsLocal) > 0
```

It ensures that only the alive node with the lowest node id in the system can trigger any of these three rules. This is done by the following set:

```
DefineSet AliveNodeWithLowestIdIsLocal :
Node System Where
LifeSignAge <= Limit, Id Min, HopCount = 0
Note "Contains the alive node with the lowest
id in the system if this is the local node"
```

The first query restricts the system nodes to the alive ones, the second query takes the one with the minimum id while the third one checks the hop count from the local node to the target node. So only if the alive node with the lowest id is local, this query is true. Otherwise the set is empty. Therefore, the above sub-condition is only true on the alive node with the lowest id.

The second group of rules handles general overload situations not caused by crashes. It contains 6 rules. Also here, all these rules use at least one local sub-condition.

1. Relocate an unhealthy application (example rule mentioned above):
   - If the application's node is unhealthy but the corresponding communication channels are still healthy, a relocation of the unhealthy application is applied.
   - This rule ensures that applications affected by an unhealthy node are moved to healthier nodes.

2.&3. Data compression (second rule of the group) or tuning (third rule of the group) of the least important applications:
   - In cases where both applications and communication channels are unhealthy, reducing the data transmission (data compression) or tuning of least important applications is applied to alleviate the communication load.

4. Tuning of local unhealthy applications:
   - If there are remaining unhealthy applications which can be tuned but the communication channels are healthy, the fourth rule is triggered.
   - This rule specifically deals with local tunable applications to reduce load on the local node.
   - This rule also makes room for relocation.

5. Global tunning of unhealty applications:
   - Similar to the fourth rule, the fifth rule addresses unhealthy applications which can be tuned when the communication channels are healthy.
   - However, it encompasses global tunable applications ordered by importance.
   - This rule also makes room for relocation.

6. Pause least important applications:
   - If there are no further options for compression or tuning applications, the sixth rule pauses the least important applications to free up resources and address the limitations in capacity or performance.

The third group is an example on how to deal with scheduling issues. It consists of 4 rules, which change scheduling and preemption policies for computing nodes and communication channels. All four rules trigger only if there are local unhealthy applications.

1.&2. Activate Priority-Based Scheduling:
   - The first two rules examine whether priority-based scheduling for nodes (rule 1) and communication channels (rule 2) is available but not currently used.
   - If priority-based scheduling is not active, this rules triggers the activation of priority-based scheduling for nodes and communication channels.
   - Application priorities are assigned based on rate monotonic scheduling (RMS) principles, ensuring efficient task scheduling.

3.&4. Activate Preemption:
   - The third and fourth rule checks whether preemption for nodes (rule 3) and communication channels (rule 4) is available but not currently utilized.
   - If preemption is not active, this rule triggers the activation of preemption capabilities for nodes and communication channels.

The fourth group of rules enables undoing of overload measures if the overall system condition improves. All three rules act on local applications.

1. Undo pausing of the most important local application:
   - The first rule focuses on undoing the pausing of the most important paused local application.
   - It applies when the health of the local node and all local communication channels surpasses a specified threshold.

2. Undo tuning of the most important local application:
   - The second rule addresses the successive undoing of tuning measures applied to the most important tuned local application.
   - It comes into effect when the health of the local node and all local communication channels exceeds a predetermined threshold.

3. Undo data compression for the most important local application:
   - The third rule focuses on undoing data compression measures applied to the most important compressed local application.

– It comes into effect when the health of the local node and all local communication channels exceeds a predetermined threshold.

The fifth group consists of a single rule which focuses to kill foreign orphans (cf. following set definition) if the connection to a CPS in the CPN is lost. The rules in the previous groups might relocate some applications to another CPS if the resources in the own CPS become critical. In case the connection to a CPS is lost, two things have to happen: first these applications have to be reassigned to the own or another CPS which is still connected. This is done by the rules of the first group, because due to the loss of connection these applications are considered as crashed. Second, their might be orphans left in the CPN, which belong to the lost CPS. These orphans have to be killed which is done by the rule in the fifth group. The following set definition contains all such foreign orphans on a local node:

```
DefineSet ForeignOrphans :
App LocalNonSystem Where SystemLifeSignAge > Limit
Note "Non-system apps running on our locale node
where we have lost contact to their systems"
```

The scope `LocalNonSystem` specifies all local applications which do not belong to the own CPS while the query tests, if the life sign age of the CPS the application belongs to has expired. Using this set, each node can kill its local orphans.

Overall, this is a basic generic rule set. It is not considered to be complete or the best possible rule set, especially when it comes to application specific behavior. However, the rules are well motivated and can be seen as a suitable basis for the following evaluations and a foundation of learning and extensions.

7.3.3. Evaluation 1 - Handling of Dynamic Load Changes

The first evaluation of the adaptation mechanism uses a single vehicle (CPS). To evaluate the potential of the adaptation mechanism, several events have been placed to change the overall load of the system. Those are summed up in Table 7.4. Thereby, the first two columns of the table show the time of the event and the event itself. First, the system load is gradually increased: After 10 seconds the performance of computing $Node_A$ is slowed down from 10 Mips to 7.5 Mips due to thermal issues. At 25 seconds the *Cruise Control* application is started by the driver. The *Infotainment* system and the *Navigation* are started after 40 seconds respectively 55 seconds. Beginning with second 70 the load now is gradually decreased back to its initial value. By this, it shall be evaluated how the adaptation mechanism not only reacts on increased but also on decreased load. Therefore, at 80 seconds the original performance of $Node_A$ is restored. Afterwards, the *Cruise Control*, *Infotainment* and *Navigation* are stopped at 80, 90 and 100 seconds.

time [s]	**Event**	**Actions (*Chameleon* instance)**	**Components**	**Fitness (before action)**	**Fitness (after action)**	**Reward (measured)**
10	Slow down $Node_A$ (10 to 7.5 Mips)					
10.3		A1: `RelocateLocalUnhealtyApp` (A)	*Stability and Drive Dynamics*: $Node_D$	0.347946	0.627541	0.279595
25	Start *Cruise Control* ($Node_C$)					
25.41		A2: `TuneLeastImportantApp` (C)	*Driver Warning*: Factor 2	0.283161	0.436721	0.15356
27.81		A3: `TuneLeastImportantApp` (C)	*Cruise Control*: Factor 2	0.327877	0.569542	0.241665
40	Start *Infotainment* ($Node_D$)					
40.25		A4: `CompressLeastImportantApp` (D)	*Infotainment*: Factor 0.8	0.369106	0.554585	0.185479
55	Start *Navigation* ($Node_D$)					
55.55		A5: `CompressLeastImportantApp` (D)	*Navigation:* Factor 0.8	0.307556	0.312089	0.004533
57.65		A6: `TuneLocalUnhealthyApps` (D)	*Navigation:* Factor 5	0.313027	0.44987	0.136843
59.65		A7: `RelocateLocalUnhealthyApp` (D)	*Navigation*: $Node_A$	0.44987	0.512715	0.062845
70	Speed up $Node_A$ (7.5 to 10 Mips)					
80	Stop *Cruise Control*					
90	Stop *Infotainment*					
90.26		A8: `UntuneMostImportantLocalApp` (A)	*Navigation*: Factor 4.5	0.59891	0.632326	0.033416
		A9: `UntuneMostImportantLocalApp` (B)	*Driver Warning*: Factor 1.5	0.598827	0.631418	0.032591
92.27		A10: `UntuneMostImportantLocalApp` (A)	*Navigation*: Factor 4	0.632326	0.647241	0.014915
		A11: `UntuneMostImportantLocalApp` (B)	*Driver Warning*: Factor 1	0.631419	0.637067	0.005648
94.28		A12: `UntuneMostImportantLocalApp` (A)	*Navigation*: Factor 3.5	0.647241	0.640076	-0.007165
96.28		A13: `UntuneMostImportantLocalApp` (A)	*Navigation*: Factor 3	0.640076	0.639025	-0.001051
98.28		A14: `UntuneMostImportantLocalApp` (A)	*Navigation*: Factor 2.5	0.639025	0.654917	0.015892
100	Stop *Navigation*					

Table 7.4: Evaluation 1 - Events and adaptation actions.

The remaining columns of Table 7.4 show the actions selected by the autonomous adaptation mechanism, the *Chameleon* middleware instance initiating the action, the affected components, the system fitness and the gained reward. Selected adaptation actions are described in more detail while explaining the evaluation results.

Figure 7.7 compares the number of unhealthy applications (applications which violate their constraints) with and without the actions of the adaptation mechanism. As well,

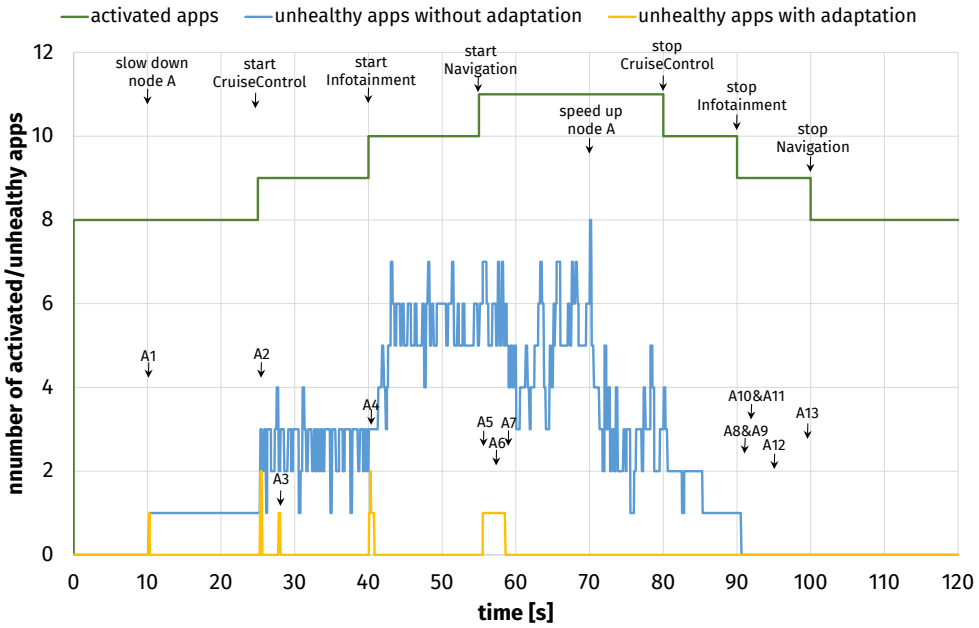


Figure 7.7: Evaluation 1 - Comparison of unhealthy applications with and without adaptation.

the number of the overall activated applications, the events and the adaptation actions as reference to Table 7.4 column three (e.g. A1 ≈ `RelocateLocalUnhealthyApp`) are shown. It can be seen that the adaptation mechanism quickly reacts to unhealthy applications in the system. Without adaptation, a large number of unhealthy applications occur. It also can be seen that some of the events can be counteracted by a single adaptation action (e.g. A1, A4) while others require a sequence of adaptation actions (e.g. A2-A3, A5-A7). None of these actions pauses an application. So here the number of actually running applications is equal to the number of activated applications (applications that want to run). The last actions (A8-A13) react to the decreased load by undoing previous adaptation measures. This is not directly reflected in the number of unhealthy apps but can be noted in Figure 7.8 where the system fitness is shown.

It can be seen that without adaptation the system fitness dramatically drops while the fitness with adaptation stays always above zero and close to the maximal possible fitness value of 1. Once the original load is gradually restored, the system fitness without adaptation recovers[1]. With adaptation, the decrease of the fitness is counteracted by

[1]To avoid thrashing, the rule engine uses the worst of the last 5 health values of a component.

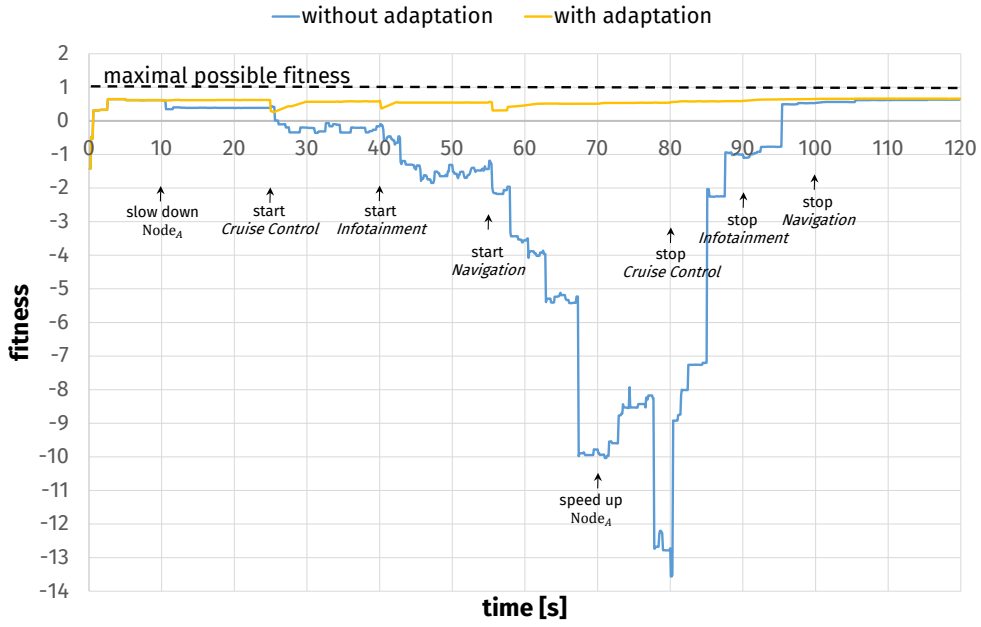


Figure 7.8: Evaluation 1 - Comparison of the system fitness with and without adaptation.

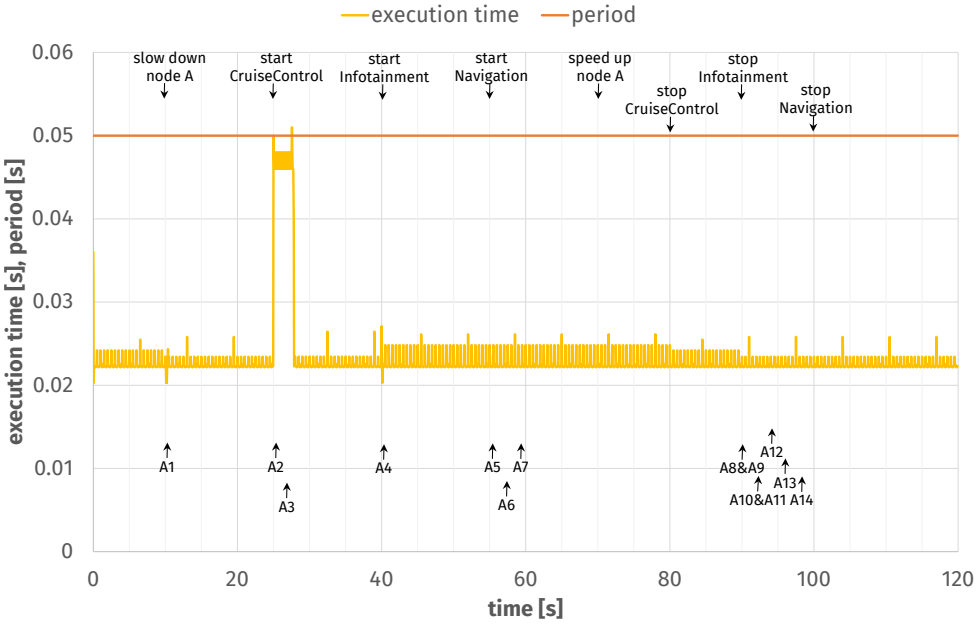
the adaptation mechanism. For example, after slowing down $Node_A$ the fitness of the non-adapted system drops while the fitness of the adapted system remains at the same level. This can be observed as well for the other events increasing the system load. When the system gets back to its initial state, the undoing of the previous adaptation measures also restore the fitness value back to its original level (like without adaptations).

The following figures show the periods, execution times and health values of selected applications in detail, ordered by their importance. First the diagrams for the most important applications (*importance* level 7) *Brake* and *Steering* are displayed.

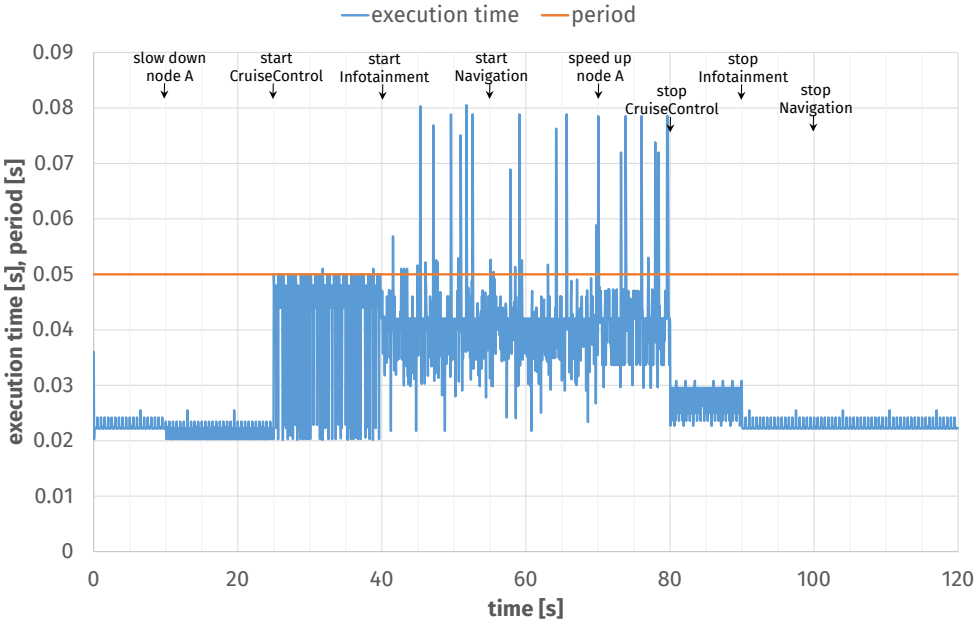
For the *Brake* (Figure 7.9), the start of the *Cruise Control* leads to a slight violation of the execution time constraint given by the period as shown in the red line. This is due to the fact that both applications reside on $Node_C$. The adaptation mechanism reacts to this violation by tuning the least important active applications in the system to the allowed limits (actions A2 and A3, cf. also Max Period (tune) in Table 7.2).

This leads to the restoration of the initial valid execution times. Also, all the other load changes can be noticed in the diagram but do not cause any further violation of the constraint. In contrast, without adaptation the constraint is heavily violated especially after the start of the *Infotainment*. This is also reflected when comparing the health values of the *Brake* application with and without adaptation as shown in

This enables a direct reaction to decreasing health values while the reaction to increasing health values is delayed. Since the fitness is based on these health values, a delayed reaction of the fitness to system improvement can be observed in the diagram.



(a) With adaptation.



(b) Without adaptation.

Figure 7.9: Evaluation 1 - Execution time and period of the *Brake* application.

Figure 7.10. In particular, the two negative peaks of the yellow health curve (with adaptation) trigger the adaptation actions A2 and A3.

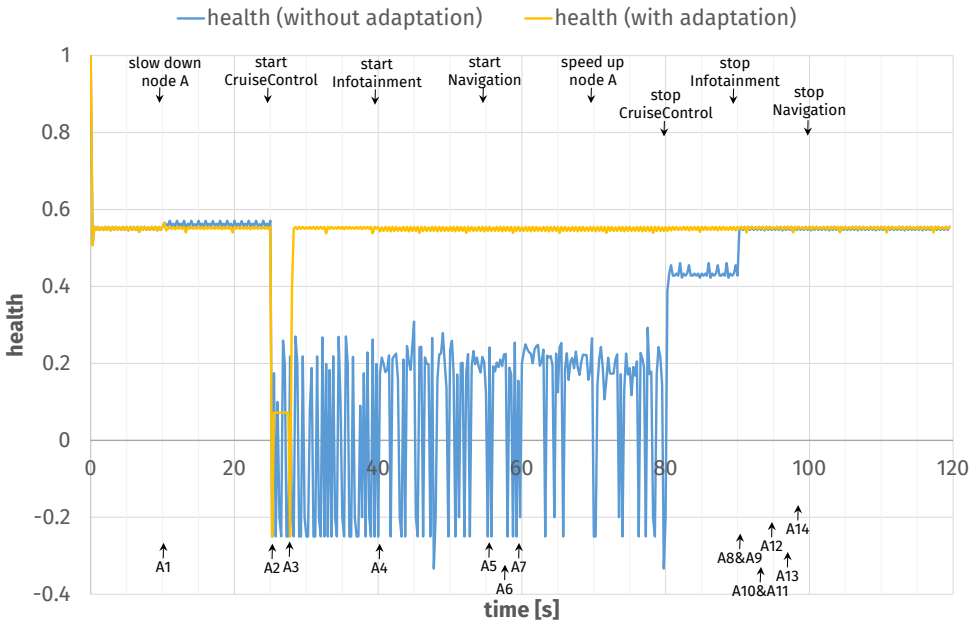
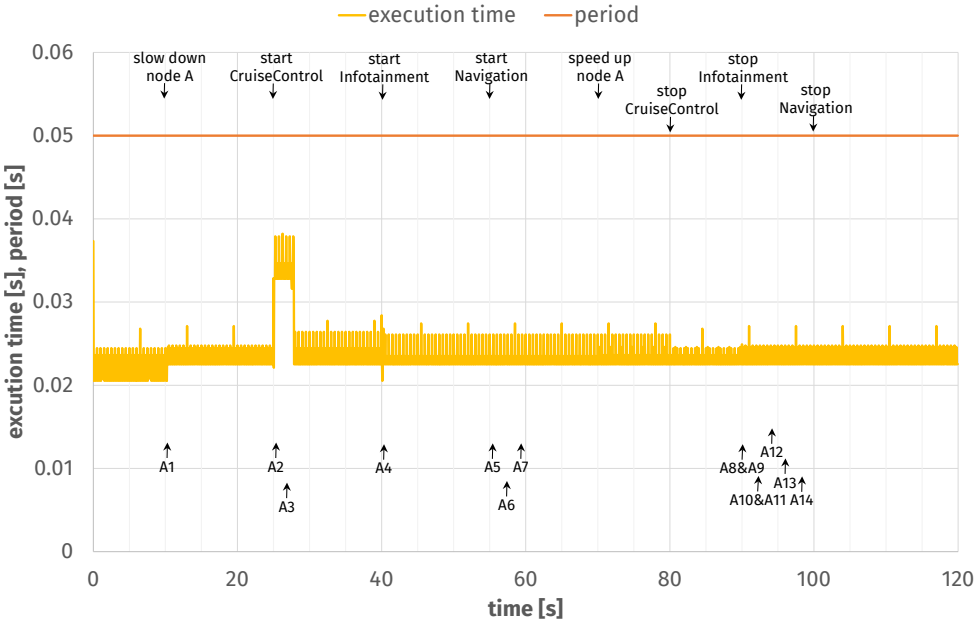


Figure 7.10: Evaluation 1: Health values of the *Brake* application.

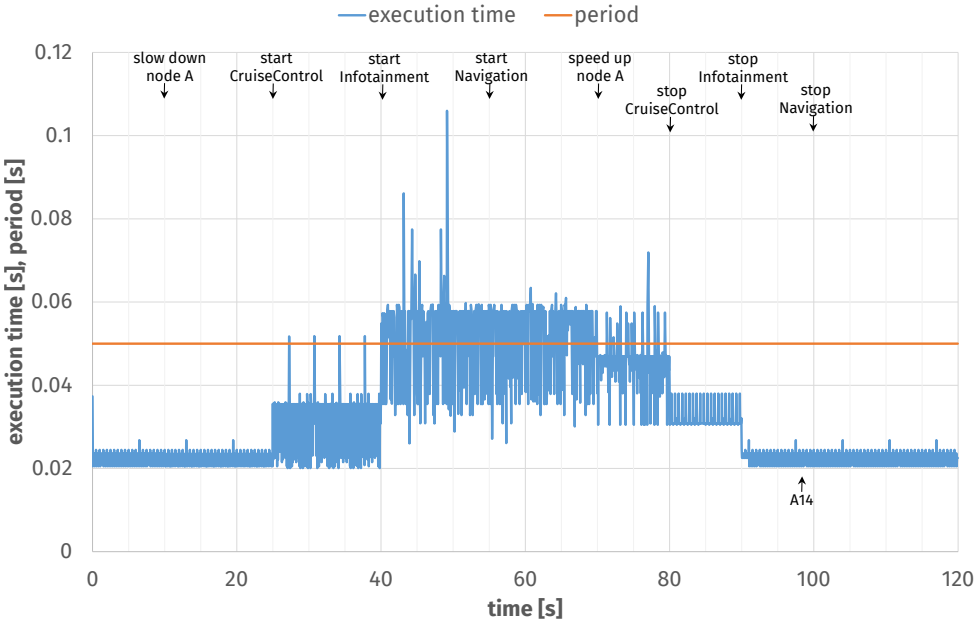
For *Steering*, no violation occur at all when the adaptation mechanism is active (cf. Figure 7.11a). In contrast, without adaptation (cf. 7.11b) the timing constraint given by the period is violated after start of the *Cruise Control* and even more after the start of *Infotainment*. The health values in Figure 7.12 clearly show this.

Next, the behavior of the medium important application *Stability And Drive Dynamics* (*importance* Level 5) is investigated. Figure 7.13 illustrates the execution time and period of this application. Due to the slow down of the performance of $Node_A$, this node no longer can handle the requirements of both applications *Powertrain* and *Stability and Drive Dynamics*. Therefore, the adaption mechanism relocates the lower important application *Stability and Drive Dynamics* to another suitable node (action A1). The effect of this relocation on can also be seen on the node load shown in Figure 7.14, where the load of the nodes $Node_A$ and $Node_D$ is illustrated. When starting *Infotainment*, another violation occurs which is counteracted by data compression of the *Infotainment* application (action A4). Without adaptation, the system gets massively overloaded leading to heavily violations of the constraint starting from the first event. This is only resolved when the performance of $Node_A$ is restored to its original value. Also, the comparison of the health values as shown in Figure 7.15 confirm this observation.

Finally, the detailed timing of the low important *Navigation* application (*importance* level 2) is shown in Figure 7.16. Once the *Navigation* is started after 55 seconds, the load of the system is even more increased. Thus, the *Navigation* is initially not able to



(a) With adaptation.



(b) Without adaptation.

Figure 7.11: Evaluation 1 - Execution time and period of the *Steering* application.

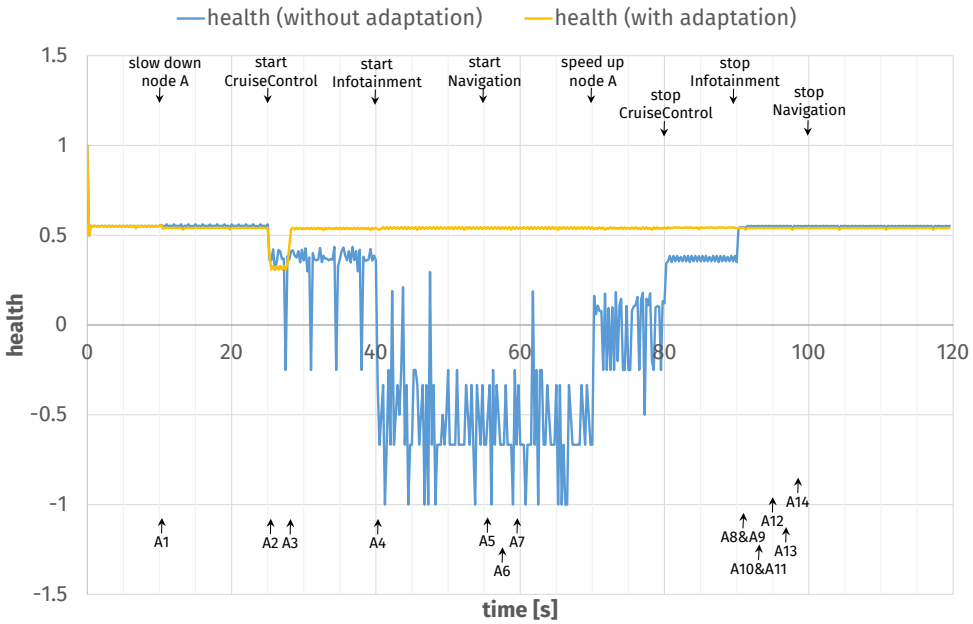
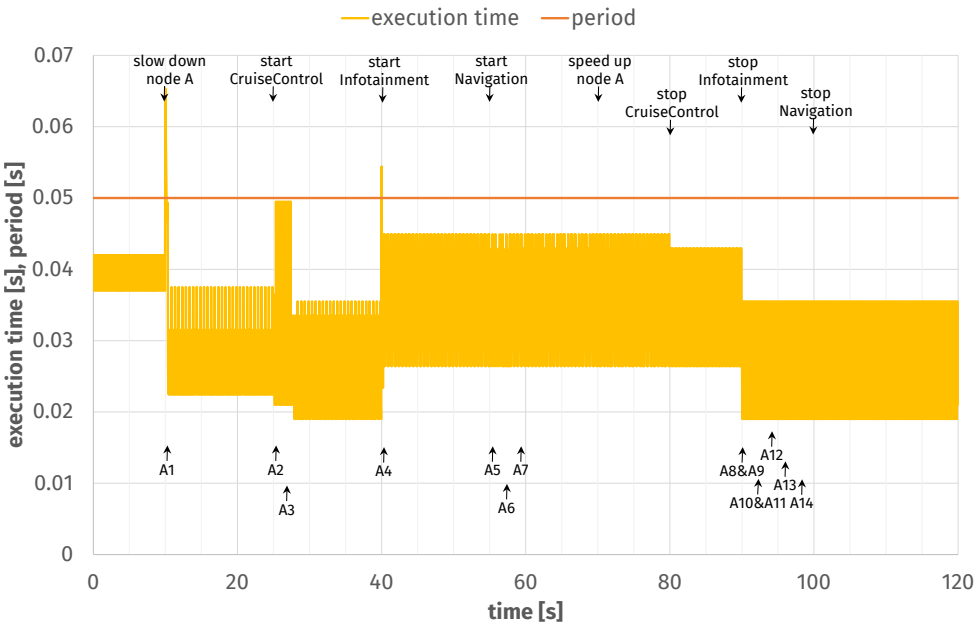


Figure 7.12: Evaluation 1 - Health values of the *Steering* application.

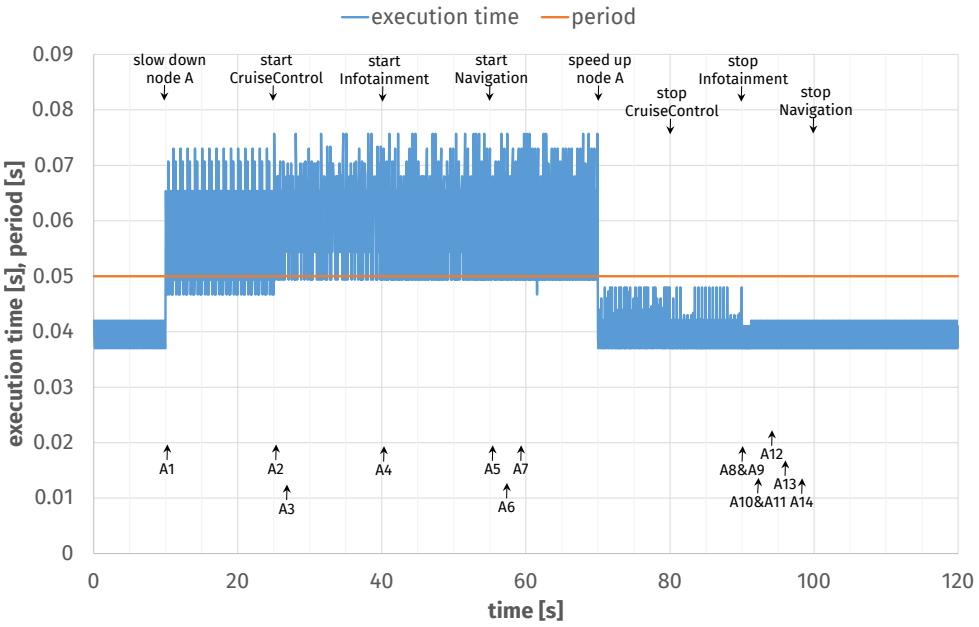
keep its timing constraints given by the period. The adaptation mechanism reacts with a sequence of three consecutive actions: First data compression (action A5) is applied. As this is not enough to solve the issue, the period of the low important application *Navigation* is tuned (action A6) to its maximum allowed value (tolerance parameter). Additionally, the adaptation mechanism initiates a relocation of the *Navigation* to reduce the load of $Node_D$ (action A7). Also, this figure demonstrates nicely the undoing of adaptation measures (untuning) when the system load is gradually decreased again. After stopping *Infotainment* by the driver, the adaptation mechanism notices the increased health of the available resources (node and bus) and therefore gradually revokes the previous tuning (actions A8, A10, A12, A13, A14)[1]. Without adaptation, the *Navigation* is completely unable to keep its constraint even when referring to the maximum allowed tune value for the period (0.5s). The comparison of the health values shown in Figure 7.17 also indicate the massive unhealthiness of the system without adaptation while with adaptation the *Navigation* is restored to healthiness within a short period of time after its activation.

The reaction times to the events comply to the bounds derived in Section 7.2. All actions are triggered on the state change of a local components (cf. Section 7.3.2), so equation 7.8 ($t_{ReactionTime_{local_j}} \leq t_{\texttt{EvaluationPeriod}} + (n_{monitorPeriods}+1) \cdot t_{MonitoringPeriod_{local_j}}$) applies for the reaction times. The involved components have a maximum local monitoring period of $250ms$ and the events increasing the load can be monitored within the duration of a single period ($n_{monitoringPeriods} = 1$). The reaction time bound then

[1]The actions A9 and A11 refer to the untuning of the *Driver Warning* application.



(a) With adaptation.



(b) Without adaptation.

Figure 7.13: Evaluation 1 - Execution time and period of the *Stability and Drive Dynamics* application.

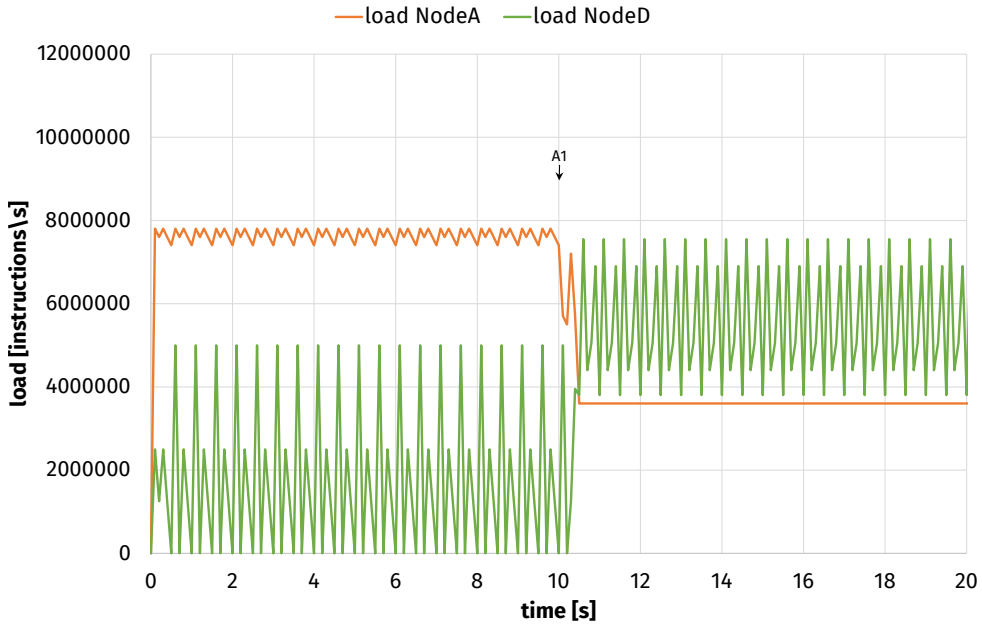


Figure 7.14: Evaluation 1 - Load of the nodes $Node_A$ and $Node_D$ (with adaptation).

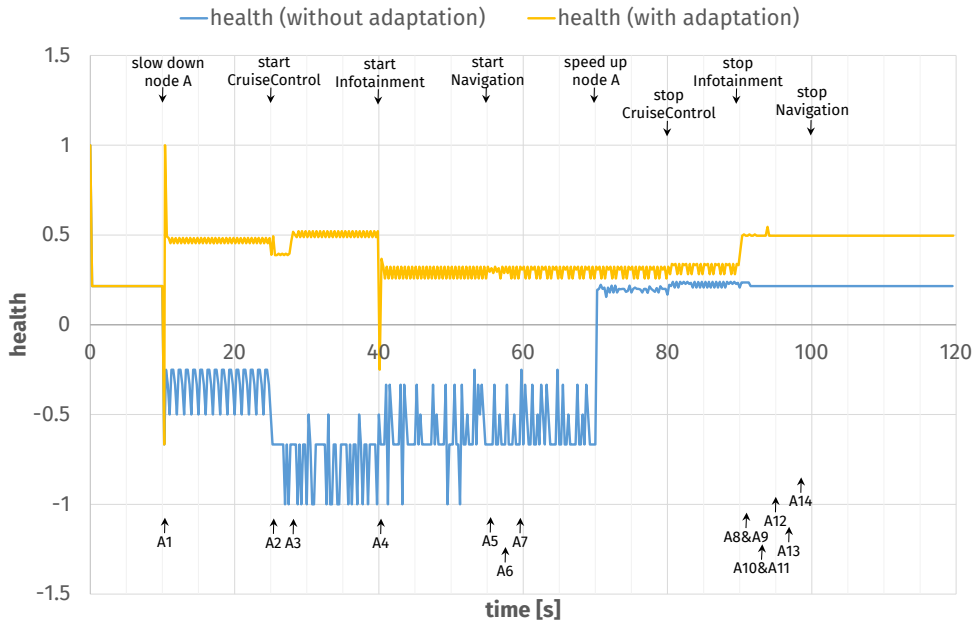
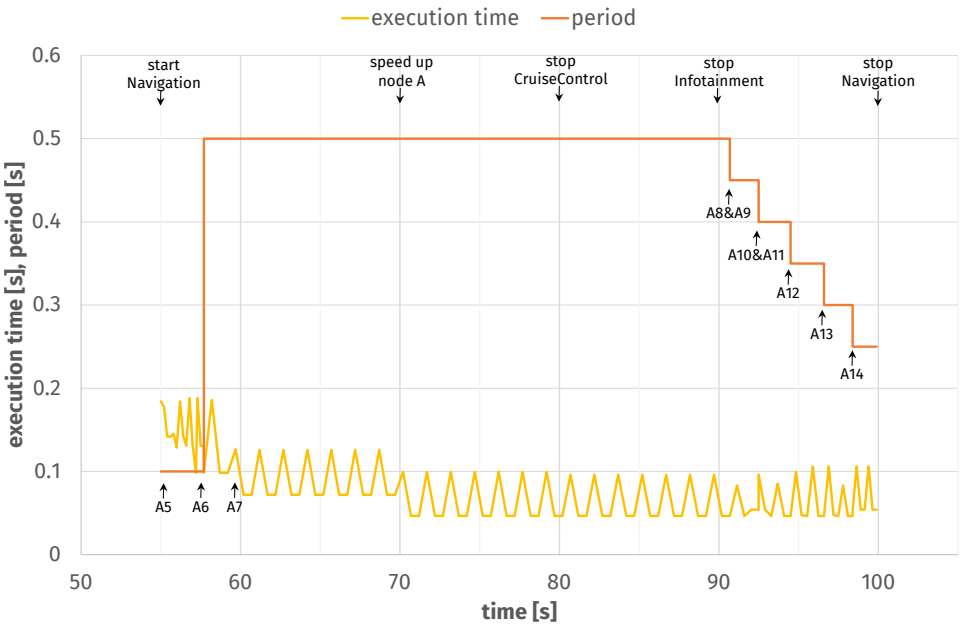
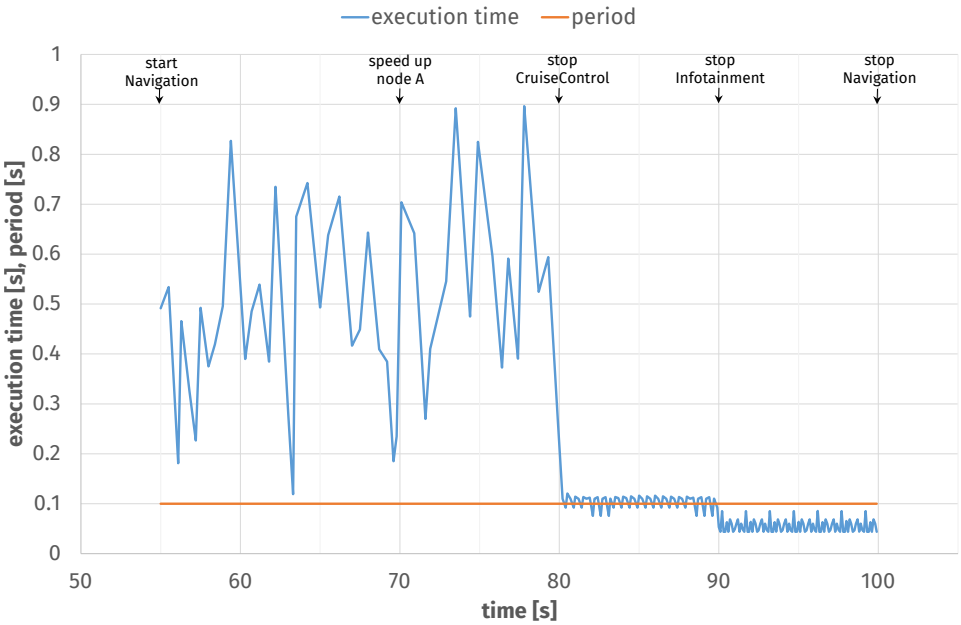


Figure 7.15: Evaluation 1 - Health values of the *Stability and Drive Dynamics* application.



(a) With adaptation.



(b) Without adaptation.

Figure 7.16: Evaluation 1 - Execution time and period of the *Navigation* application.

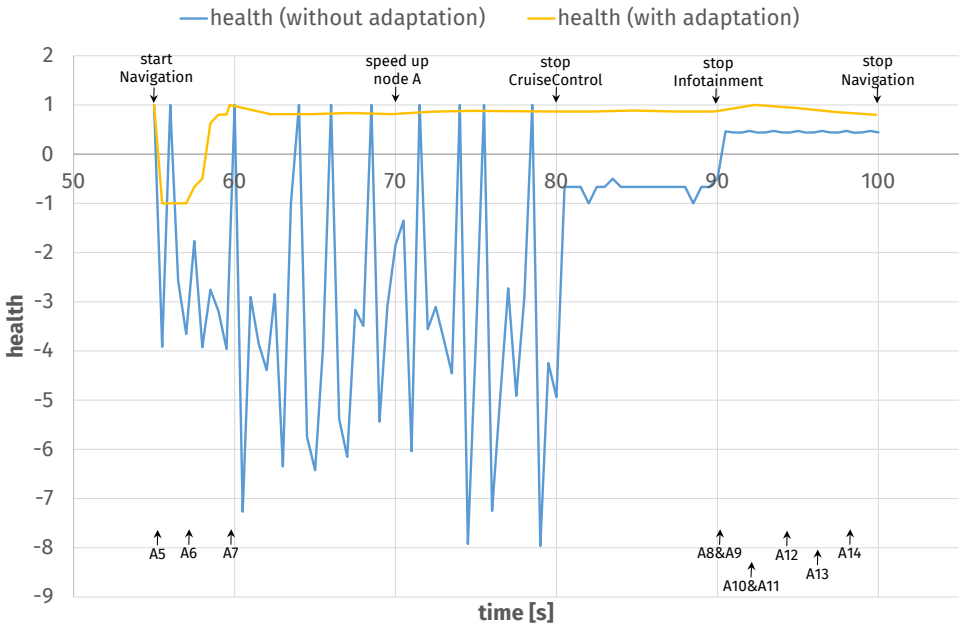


Figure 7.17: Evaluation 1 - Health values of the *Navigation* application.

calculates to $100+2\cdot250 = 600ms$. This bound is kept for all reaction times in Table 7.4 ($10s \rightarrow 10.3s,\ 25s \rightarrow 25.41s,\ 40s \rightarrow 40.25s,\ 55s \rightarrow 55.55s,\ 90s \rightarrow 90.26s$). For the sequences of actions A5-A7 and A8-A14, the adaptation times according to equation 7.21 ($t_{Adaptation} \leq t_{ReactionTime_1} + \sum_{i=1}^{x-1}(t_{\texttt{RewardDelay}_i} + (\texttt{Lag}_{i+1} - 1) \cdot t_{\texttt{EvaluationPeriod}})$) also hold. The reward delay for all this actions is the global value of $2000ms$ and the `Lag` is 1 for all actions except A6, which has a value of 2 (cf. Appendix B.5). Furthermore, the sequence A8-A14 is partially executed on different *Chameleon* instances, so a small communication overhead can be noticed. However, the behavior of A2-A3 is different because this is not a real sequence but two separate actions. This can be clearly seen in Figure 7.10, A2 is triggered by the unhealthy *Brake* after starting *Cruise Control*, which rises the health value marginally above 0 (0.072) and therefore finishes the adaptation process. However, due to variations in the execution of *Cruise Control*, the health value of *Brake* again gets negative 2.7 seconds later triggering the second adaptation A3. This adaptation now solves the problem and increases the health value of Brake to a stable level above 0.5. So, A2 and A3 are separately triggered actions, which both keep the reaction time bound of $600ms$ after the system gets unhealthy.

To get an impression of the communication situation during the experiment, Figure 7.18 examines the load of the communication bus. Once the additional applications are started, the bus becomes a bottleneck due to the additional communication load. The maximum capacity of the bus (500000 bits/s) (cf. Section 7.3.1) is strongly exceeded without adaptation. With adaptation, the load mostly gets aligned to or below its maximum value.

The previous diagrams also demonstrate that the health values are an excellent

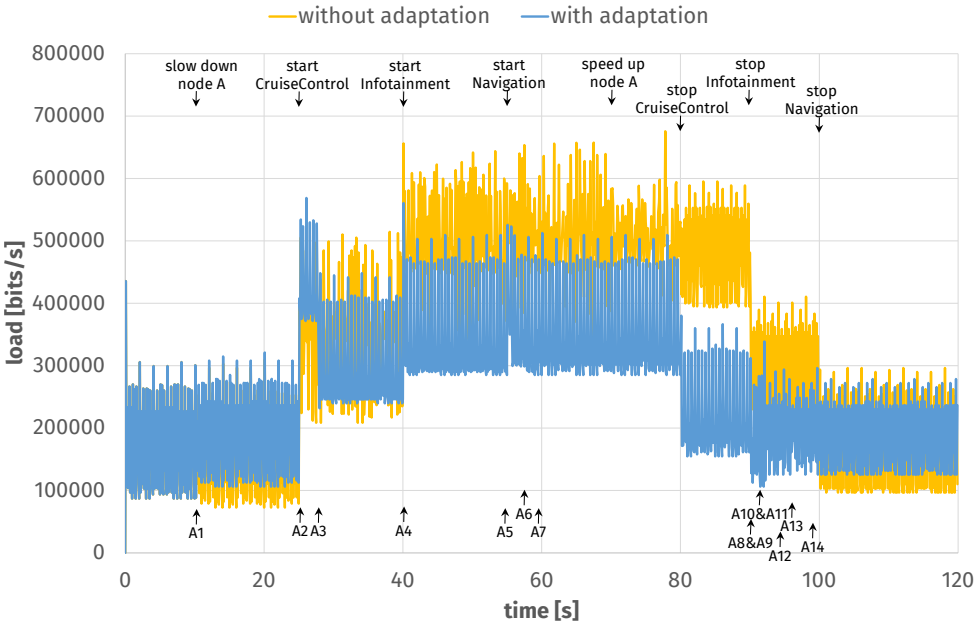


Figure 7.18: Evaluation 1 - Load of the communication bus $Comm_1$.

indicator for the violation of constraints and impact of the adaptation actions. Thus, in the following evaluations the health values are used as the main metric for the adaptation results.

Conclusion This evaluation clearly shows the benefits of the adaptation mechanism. It is able to keep the system in a healthy state by keeping the defined constraints most of the time. Especially for the high important apps the violations are non existent to marginal. Without adaptation, the figures show a huge amount of violations. It also can be seen that mixed-criticality is respected by the adaptation process due to the basic rule set. For example, regarding tuning and compression the least important applications are affected to create capacity for the more important applications. The evaluation also confirms the time bounds retrieved in Section 7.2.

In following evaluation sections, these findings for the adaptation mechanism will be successively deepened and extended.

7.3.4. Evaluation 2 - Autonomous Versus Manual Adaptation

Before the implementation of the adaptation mechanism was finished, there was the idea to test the effectivity of the available adaptation actions manually as a proof of concept. The goal was to determine if the chosen action portfolio (cf. 4.4) would be suitable to handle overload situations. The same application scenario used for evaluation 1 including all events up to second 70 were chosen for this early experiment[1]. As reaction to the events, adaptation actions were carefully selected manually by the application developer. Besides its original goal of validating the effectivity of the action portfolio, this experiment now also can serve as a comparison between the effectivity of the autonomous adaptation mechanism versus manual adaptation by a human developer. Although not being a strict evidence, it gives a good impression on the quality and capability of the selected autonomous self adaptation mechanism in conjunction with its rule set. Table 7.5 shows the manually chosen adaptation actions in comparison to the autonomously selected ones.

time [s]	Event	Actions (*Chameleon*)	Action (manual)	Components
10	Slow down $Node_A$ (10 to 7.5 Mips)			
10.3		`RelocateLocalUnhealtyApp` (A1)		*Stability and Drive Dynamics*: $Node_D$
15			Relocate	*Stability and Drive Dynamics*: $Node_D$
25	Start *CruiseControl* ($Node_C$)			
25.41		`TuneLeastImportantApp` (A2)		*Driver Warning*: Factor 2
27.81		`TuneLeastImportantApp` (A3)		*Cruise Control*: Factor 2
30			Tune period and priority	*Cruise Control*: Factor 1.5
40	Start *Infotainment* ($Node_D$)			
40.25		`CompressLeastImportantApp` (A4)		*Infotainment*: Factor 0.8
45			Compress	*Infotainment*: Factor 0.75
			Tune period and priority	*Infotainment*: Factor 2
55	Start *Navigation* ($Node_D$)			
55.55		`CompressLeastImportantApp` (A5)		*Navigation:* Factor 0.8
57.65		`TuneLocalUnhealthyApps` (A6)		*Navigation:* Factor 5
59.65		`RelocateLocalUnhealthyApp` (A7)		*Navigation*: $Node_A$
60			Pause	*Infotainment*
			Tune period and priority	*Navigation*: Factor 3

Table 7.5: Evaluation 2 - Events and adaptation actions of autonomous and manual adaptation.

The manual actions had been chosen by the developer based on the node and communication load. While the first action (action A1: relocate application to a suitable node) is identical for both manual and autonomous adaptation, in the following the autonomous actions are more fine grain and less restrictive. In particular, the autonomous adaptation succeeds to keep the system healthy without pausing any application by efficiently combining compression, tuning and relocation (e.g. actions A5 - A7). The human developer did not find this solution and paused one application

[1]The manual adaptation experiment focused on increasing load only. The subsequent decrease of load beginning with second 70 was added later for evaluation 1.

instead. Overall, the autonomous adaptation performs significantly better than the manual adaptation, as also can bee seen in the number of unhealthy applications compared in Figure 7.19. Also, the pausing of an application by the manual adaptation can be seen in the number of running applications[1].

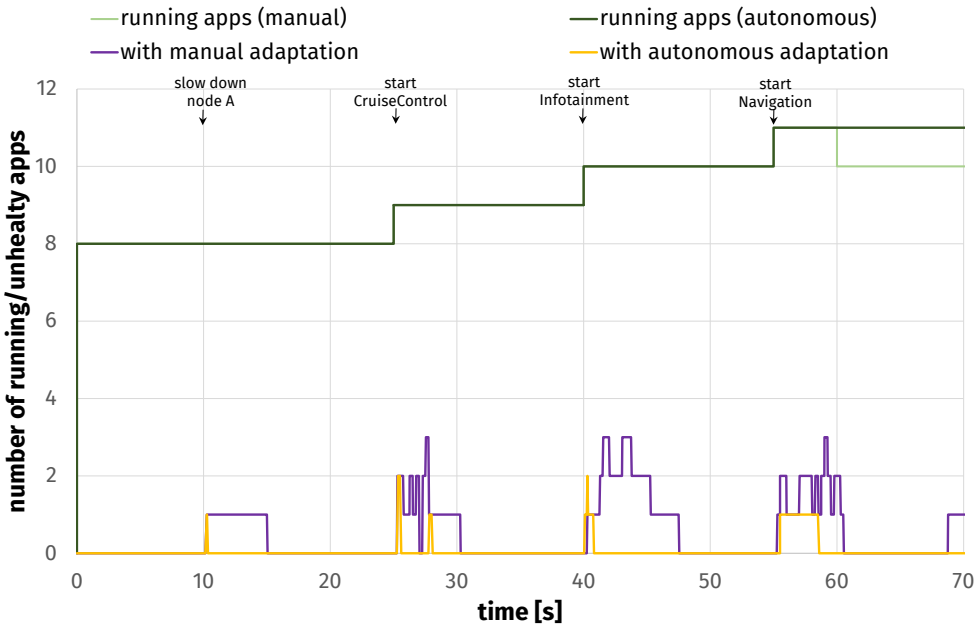


Figure 7.19: Evaluation 2 - Comparison of unhealthy applications with autonomous and manual adaptation including the number of running applications.

Conclusion Autonomous adaptation is a good choice for maintaining the system in a healthy state and is able to find solutions hardly to see by a human.

7.3.5. Evaluation 3 - Communication Overhead

This evaluation examines the communication overhead introduced by the *Chameleon* middleware. The overhead refers to the additional communication data required to support the management and adaptation processes performed by *Chameleon*. It results mainly from the data for the overlay network routing of the middleware (where is which application running and how can it be reached) and the data of the remote monitoring[2] (cf. Sections 4.1, 4.3 and 5.1).

[1] Activated application: An application which should be running; Running application: An application which is running.

[2] The communication overhead depends on the period and the number of events (changes) that lead to the distribution of parameter and status data. The more frequently the data is distributed in the system, the more up-to-date the remote monitoring data is but the higher the communication overhead for the dissemination will become. Remote monitoring data is disseminated each n^{th} local monitoring period, where n has been set to 2 for all evaluations.

Figure 7.20 shows the overhead percentage ($overhead_{[\%]}$) during execution of evaluation 1. It indicates the additional communication effort in relation to the payload. In this context, payload ($payload_{[bits/s]}$) refers to the data being transmitted by the applications to perform their tasks. It represents the essential bits per second that are conveyed and processed. The overhead ($overhead_{[bits/s]}$) refers to the additional communication data in bits per second introduced as a result of the management and adaptation processes performed by the *Chameleon* middleware.

Thus, the overhead percentage $overhead_{[\%]}$ is calculated as follows:

$$overhead_{[\%]} = \begin{cases} 0 & \text{if } payload_{[bits/s]} + overhead_{[bits/s]} = 0 \\ \frac{overhead_{[bits/s]} \cdot 100}{payload_{[bits/s]} + overhead_{[bits/s]}} & \text{else} \end{cases} \tag{7.25}$$

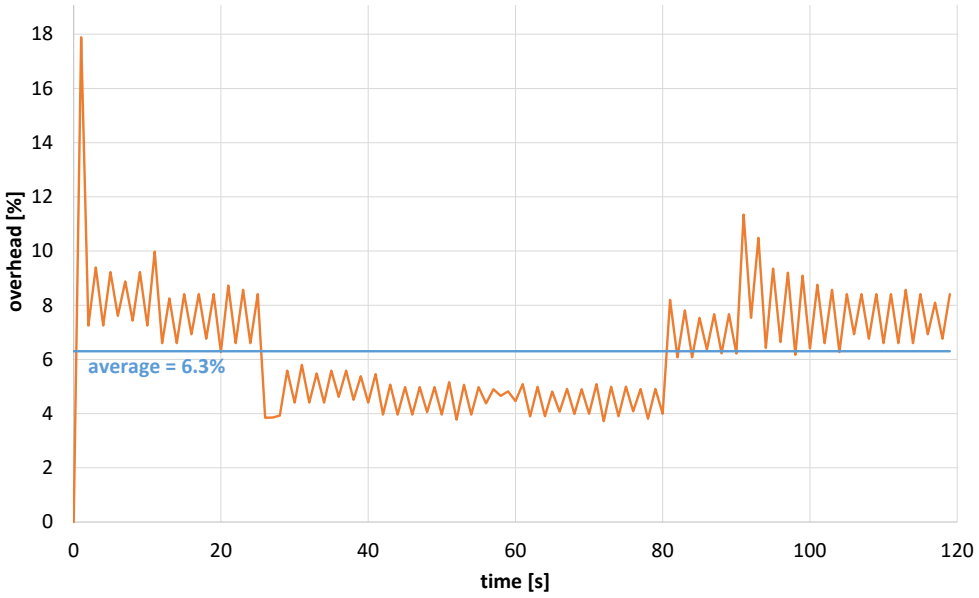


Figure 7.20: Evaluation 3 - Percentage communication overhead of the adaptive middleware *Chameleon*.

At the beginning, the overhead percentage is relatively high with 17.9%. This is the case since the payload is rather small at system startup because the applications are just starting. However, there is an overhead due to the notification events for setting up the overlay network. Once the system has settled, the overhead is much lower and relatively constant. The average overhead percentage is 6.3% and the minimum is 3.7%. The fluctuations in the overhead percentage after the system startup are caused by the varying total payload in the system. In general, the absolute overhead increases linearly with the number of active applications, regardless of how much payload the applications generate. This is clearly visible in Figure 7.20. As soon as (data-intensive) applications

are started from second 25 (start of *Cruise Control*, *Infotainment* and *Navigation*), the absolute overhead remains almost constant while the payload increases noticeably, resulting in a lower overhead percentage. After 80 seconds, those applications are gradually stopped which leads to an increase of the overhead percentage due to the decreasing total payload in the system. This can also be seen when looking at the overall communication load already shown in Figure 7.18 of evaluation 1.

Conclusion The evaluation shows a reasonable small overhead for the communication of the adaptive middleware *Chameleon*.

7.3.6. Evaluation 4 - Handling of Failures

In the following, the pressure on the adaptation mechanism is increased by introducing node failures. A node failure causes the middleware instance and all applications running on this node to crash. It will be examined how the system reacts on such failures. In a first step, the failure of a single node is triggered. The same scenario as in evaluation 1 including all events is used. Additionally, a failure of $Node_B$ is injected after 5 seconds. Table 7.6 shows all events and the corresponding actions of the adaptation mechanism.

It can be seen in Figure 7.21 that the adaptation mechanism is capable of quickly restarting the applications crashed by the node failure.

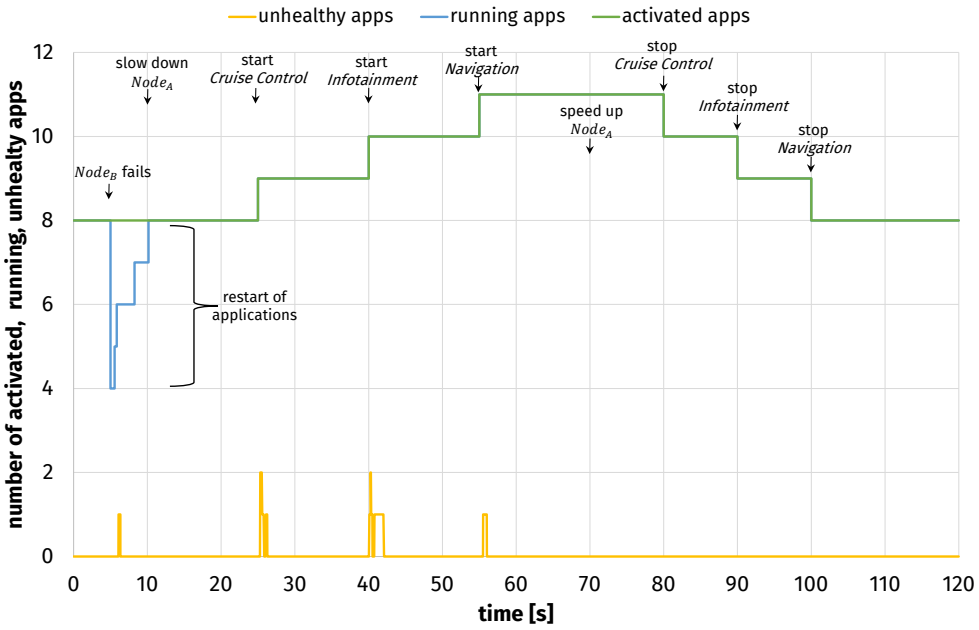


Figure 7.21: Evaluation 4 - Number of unhealthy, running and activated applications.

Furthermore, only when the load in the system is further increased there is shortly a maximum of two unhealthy applications in the system. Otherwise, the adaptation

mechanism manages to maintain all applications, including the restarted ones, in a healthy state even with only three nodes left. The good condition of the entire system despite the failure is also reflected in the fitness (cf. Figure 7.22).

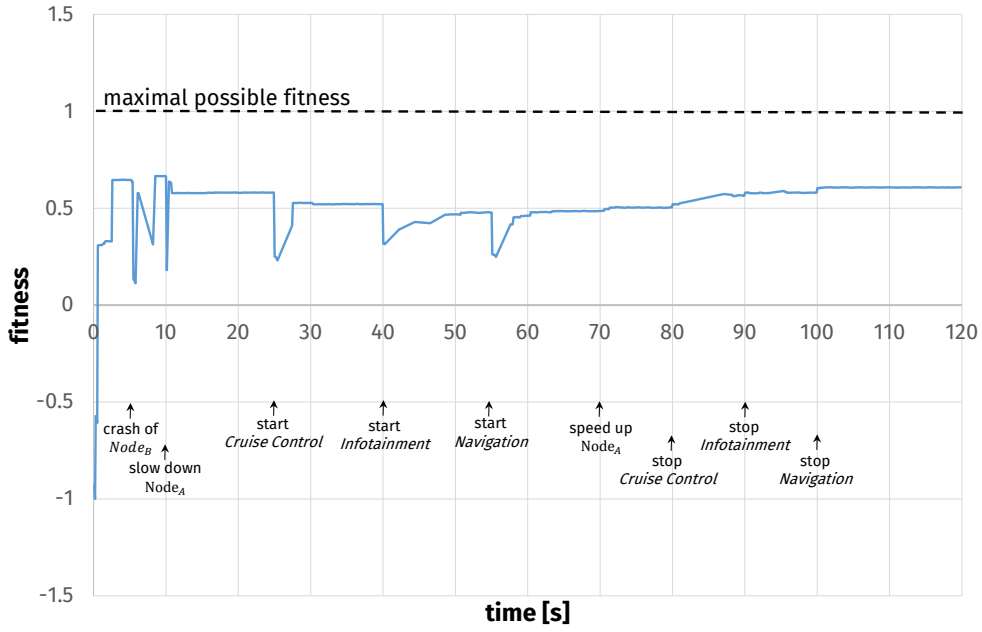


Figure 7.22: Evaluation 4 - System fitness.

Let us now have a look at the restart sequence. As expected, the two most important applications (*Passenger Safety*, *Steering*, both with *importance* level 7) are restarted first. The time bound for the reaction time here calculates according to Equation 7.12 ($t_{ReactionTime_{Crash_k}} \leq l_{LifeSignLimit} \cdot t_{MonitoringPeriod_{remote_k}} + t_{Communication_k} + t_{\texttt{EvaluationPeriod}}$). Since the remote monitoring period for *Passenger Safety* is $250ms$[1] and the life sign limit $l_{LifeSignLimit}$ is 2, the reaction time bound calculates to $600ms$ plus the communication time. This bound is kept for restarting *Passenger Safety* (A1) as can be seen in Table 7.6. The rule for restarting a crashed application has an individual reward delay of $300ms$ (cf. action definition in Section B.4). Therefore, as second action of the sequence the application *Steering* is restarted this time later (A2). Then, again after $300ms$, the application *Stability and Drive Dynamics* (*importance* level 5) is tuned (A3). Finally, the application *Driver Warning* (*importance* level 3) is restarted (A4). Since the tuning action (A3) uses the global reward delay of $2000ms$ and due to communication (A4 is executed on *Chameleon* instance A while A3 is executed on instance D), this happens $2120ms$ later (cf. Equation 7.21: $t_{Adaptation} \leq t_{ReactionTime_1} + \sum_{i=1}^{x-1}(t_{\texttt{RewardDelay}_i} + (\texttt{Lag}_{i+1} - 1) \cdot t_{\texttt{EvaluationPeriod}})$). The restart of the fourth application *Lights* does not belong to this sequence. The reason for that is not the slow down event of $Noda_A$ at 10 seconds. In fact, the reason is the

[1] 2-times its local monitoring period, which is 5-times its execution period.

remote monitoring period of *Lights*. Due to its slow execution period ($500ms$), the remote monitoring period of *Lights* is 5 seconds. Therefore, its worst case reaction time to the crash calculates to 10 seconds plus the communication time. At the time A4 is executed, the crash of *Lights* has not even been noticed yet. This also explains why the higher important application *Lights* (*importance* level 6) is restarted after the lower important application *Driver Warning*[1]. Thus, action A5 to restart *Lights* is not part of the sequence, but a separate action which keeps its reaction time bound.

Finally, when looking at the reaction times of the other actions it can be seen that the actions performed while the system load rises (A6-A10) keep the reaction time bound of $600ms$ retrieved in Section 7.3.3. However, the actions performed while the system load decreases (A12-A15) seem violate this bound. This is explained by the value of $n_{MonitoringPeriods}$ in Equation 7.8 ($t_{ReactionTime_{local_j}} \leq t_{\texttt{EvaluationPeriod}} + (n_{monitorPeriods} + 1) \cdot t_{MonitoringPeriod_{local_j}}$). As already mentioned in Section 7.2, $n_{MonitoringPeriods} = 5$ for increasing health values (decreasing load) while it is 1 for decreasing health values (increasing load). It allows to quickly react to decreasing conditions and to avoid trashing when conditions increase. In combination with the slow local monitoring period of *Lights* and *Driver Warning* ($1.25s$), the reaction time bound calculates to $7.6s$, which is easily kept.

Conclusion The adaptation mechanism is not only capable of quickly restarting the applications crashed by the node failure, but also manages to maintain the system in good condition where all applications are running and in a healthy state. Even with one node less the autonomous adaptation finds a solution to keep the complete functionality at partially reduced quality for the low important functions.

[1]Such behavior can be circumvented by using the node life sign instead of the application life sign to detect applications crashed due to a node failure, as done in evaluation 6 (c.f. Section 7.3.8). It allows to notice the crashed applications simultaneously.

time [s]	Event	Actions (*Chameleon* instance)	Components	Fitness (before action)	Fitness (after action)	Reward (measured)
5	$Node_B$ fails					
5.5		A1: `RestartDeadApp` (A)	*Passenger Safety*: $Node_D$	0.13605	0.112684	-0.023366
5.8		A2: `RestartDeadApp` (A)	*Steering*: $Node_D$	0.112685	0.57859	0.465905
6.1		A3: `TuneLeastImportantApp` (D)	*Stability and Drive Dynamics*: Factor 2	0.415416	0.313172	-0.102244
8.22		A4: `RestartDeadApp` (A)	*Driver Warning*: $Node_C$	0.313172	0.665961	0.352789
10	Slow down $Node_A$ (10 to 7.5 Mips)					
10.12		A5: `RestartDeadApp` (A)	*Lights*: $Node_C$	0.180809	0.638305	0.457496
25	Start *Cruise Control* ($Node_C$)					
25.43		A6: `TuneLocalUnhealtyApp` (C)	*Cruise Control*: Factor 2	0.249993	0.527648	0.277655
40	Start *Infotainment* ($Node_D$)					
40.25		A7: `CompressLeastImportantApp` (D)	*Infotainment*: Factor 0.8	0.384588	0.3932	0.008612
42.39		A8: `TuneLocalUnhealtyApp` (C)	*Driver Warning*:: Factor 2	0.3932	0.429646	0.036446
44.49		A9: `TuneLeastImportantApp` (C)	*Infotainment*: Factor 6	0.428949	0.422615	-0.006334
46.59		A10: `TuneLeastImportantApp` (C)	*Lights*: Factor 2	0.423491	0.466639	0.043148
55	Start *Navigation* ($Node_D$)					
55.6		A11: `TuneLocalUnhealtyApp` (D)	*Navigation*: Factor 5	0.376648	0.416008	0.03936
70	Speed up $Node_A$ (7.5 to 10 Mips)					
80	Stop *Cruise Control*					
81.03		A12: `UntuneMostImportantLocalApp` (C)	*Lights*: Factor 1.5	0.52719	0.545674	0.018484
83.03		A13: `UntuneMostImportantLocalApp` (C)	*Lights*: Factor 1.0	0.545674	0.569585	0.023911
85.03		A14: `UntuneMostImportantLocalApp` (C)	*Driver Warning*: 1.5	0.569585	0.566342	-0.003243
90	Stop *Infotainment*					
93.33		A15: `UntuneMostImportantLocalApp` (C)	*Driver Warning*: Factor 1.0	0.579286	0.584418	0.005132
100	Stop *Navigation*					

Table 7.6: Evaluation 4 - Events and adaptation actions.

7.3.7. Evaluation 5 - Handling of Failures Leading to Extreme Overload Situations

In the previous evaluation, the adaptation mechanism was able to keep all applications running even with a single node failure. Now, the pressure and load on the system is maximized by successively injecting node failures unless only a single node is left. In particular, the aspect of mixed-criticality and the influence of the *importance* can be investigated in such an extreme overload situation. The starting scenario is the same as for evaluation 1. The failure of the nodes $Node_B$, $Node_C$ and $Node_D$ are injected after 5, 20 and 35 seconds. In contrast to evaluation 1, no other events are triggered to focus on the effect of massive node failures only. Table 7.7 shows the events and the actions of the adaptation mechanism. It can be seen in Figures 7.23 and 7.24 that the adaptation mechanism is capable of quickly restarting the applications crashed by node failure as long as there is sufficient computing capacity left.

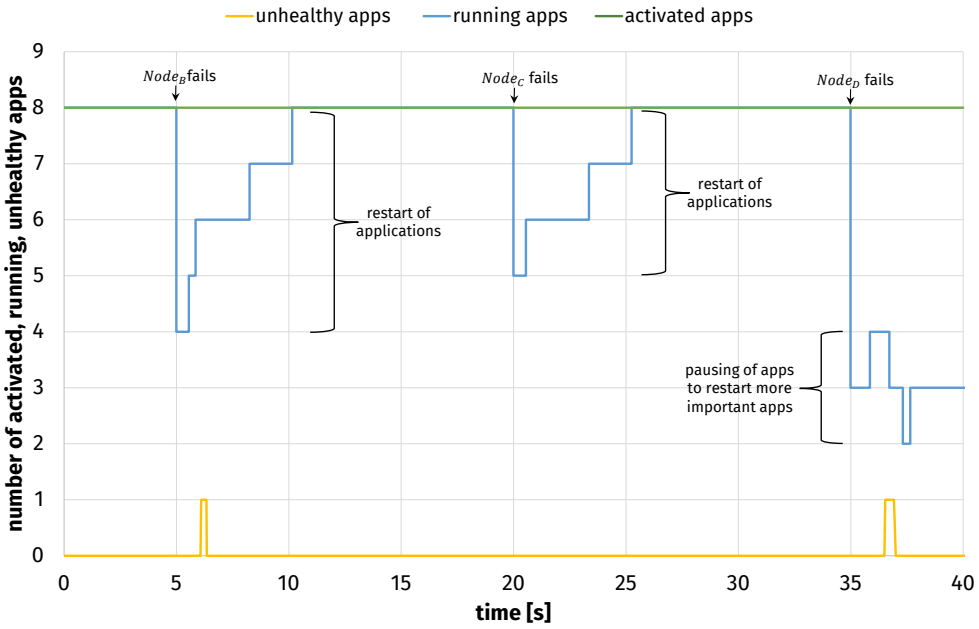


Figure 7.23: Evaluation 5 - Number of unhealthy, running and activated applications.

Figure 7.24 thereby shows the allocation of applications ordered by *importance* (highest *importance* on top).

Also it can be noticed, that as long as more than one node remains available, the adaptation mechanism manages to keep all applications running only by restarting the dead applications and tuning *Stability and Drive Dynamic* to the allowed limits (actions A1-A8). Thereby, crashed applications are restarted beginning with the highest importance. As in the previous evaluation, *Lights* (*importance* level 6) is only restarted after *Driver Warning* (*importance* level 3) due to *Lights* longer remote monitoring

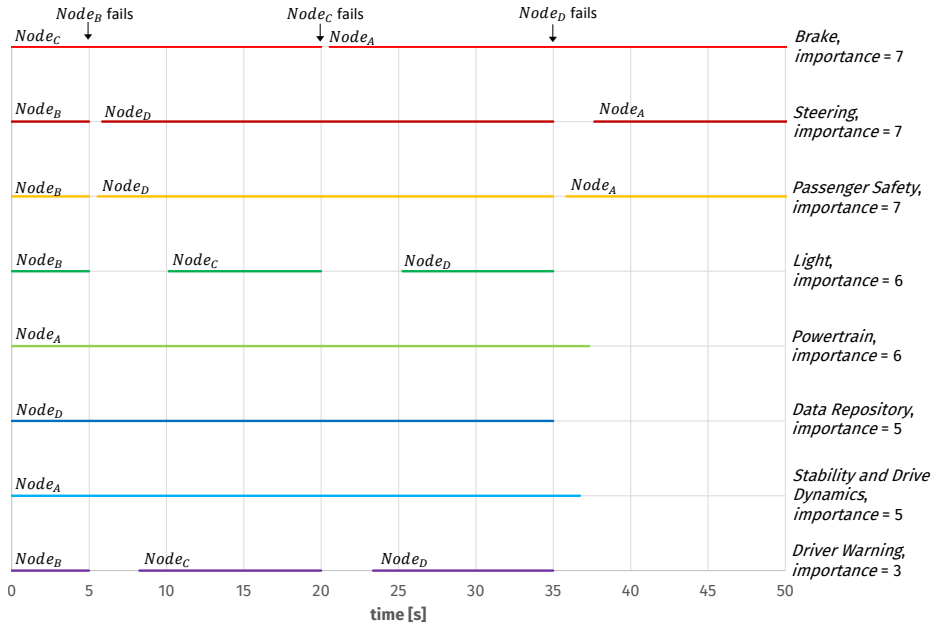


Figure 7.24: Evaluation 5 - Allocation of applications on nodes ordered by *importance*.

period which leads to a later notice of the failure (cf. Evaluation 4 7.3.6)[1].

The overall good condition of the entire system despite the failures is reflected in the fitness (cf. Figure 7.25). The fitness value drops shortly below zero when $Node_D$ fails and thus, only $Node_A$ is left for application execution. After the failure of $Node_D$ the adaptation mechanism first restarts the highest important application *Passenger Safety* (actions A9-A10)[2]. Since the system load then is to high for further recovery, the task dropping strategy pauses applications beginning with the lowest importance in strictly ascending order to be able to restart the most important application *Steering* (actions A11-A16). Figure 7.24 and Table 7.7 can be used to more detailed explain the behavior of the first group of rules (which is dedicated to handle crashes, cf. Section 7.3.2 and Appendix B.5) in the basic rule set when $Node_D$ fails.

The applications *Passenger Safety*, *Steering*, *Lights*, *Data Repository* and *Driver Warning* directly die together with $Node_D$. First, *Passenger Safety* is restarted. Then, the lowest important applications are paused to free resources. This leads to the stop of the applications *Stability and Drive Dynamics* and *Powertrain*. Pausing is done regardless if an application is crashed or not, thus ensuring the applications are paused in strictly ascending order of importance. Therefore, also the dead applications *Driver Warning*, *Data Repository* and *Lights* are paused according to their importance to

[1] As already mentioned in the previous evaluation such behavior can be circumvented by using the node life sign instead of the application life sign to detect applications crashed due to a node failure, as done in evaluation 6 (c.f. Section 7.3.8). It allows to notice the crashed applications simultaneously.

[2] Life signs of applications and nodes are sent independently of each other. It is possible that the application live sign has expired, but the node's has not. Thus, the adaptation mechanism first tries to restart *Passenger Safety* on the failed $Node_D$.

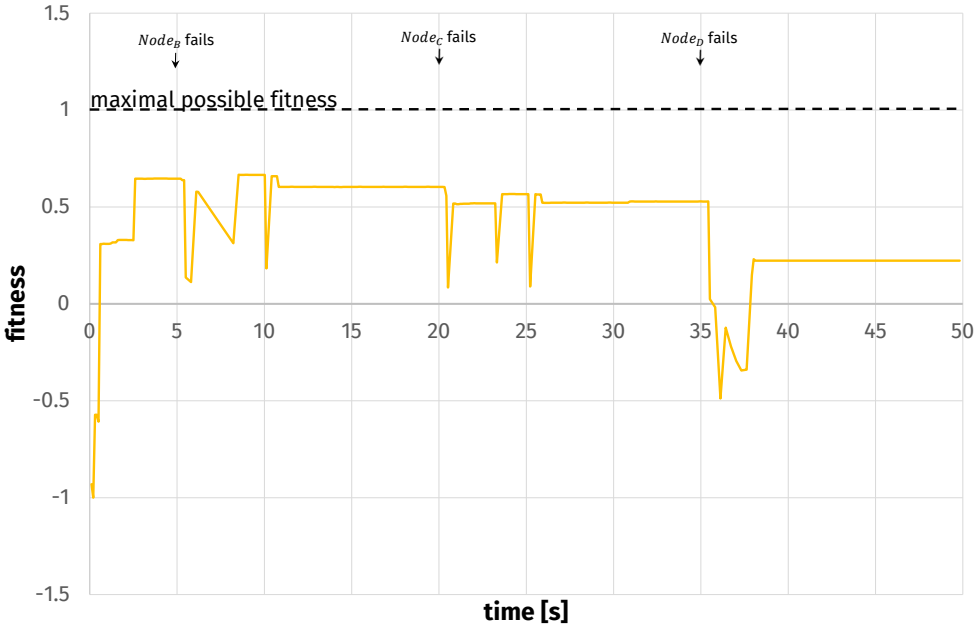


Figure 7.25: Evaluation 5 - System fitness.

prevent them from being restarted at the cost of higher important applications. Finally, the high *importance* application *Steering* is restarted. The timing constraints derived in Section 7.2 are also kept. Especially the individual reward delay of $300ms$ for restarting and pausing is directly reflected in the sequence A9 - A16, which is executed on the same *Chameleon* middleware instance. Together with the reaction time of $520ms$, the overall adaptation process takes 2.62 seconds after the failure of $Node_D$[1].

In the end, the three highest important applications (*Brake*, *Steering*, *Passenger Safety*) are running on the only remaining $Node_A$. Thus, the first group of rules has ensured that applications are kept alive in the order of importance as desired. Furthermore, the health of the most important applications is illustrated in Figure 7.26. It clearly can be seen that all applications are in a healthy state (nearly) the whole time. Only the health of *Steering* shortly drops below zero when $Node_B$ fails, on which the application is running.

Conclusion The adaptation mechanism is able to handle extreme overload situation and behaves according to the application's *importance* by tuning and or pausing of less important applications for the benefit of the higher important ones. Thus, the mixed-criticality of the system is respected by keeping the most important application alive as long as the resources are still sufficient.

[1]For a highly critical application like Steering this is a long time which has to be covered by a backup emergency mode. A possibility to speed this up is shown in the next evaluation section. It also has to be taken into account that this is an extreme overload situation where 3 out of 4 nodes are failing within a short period of time.

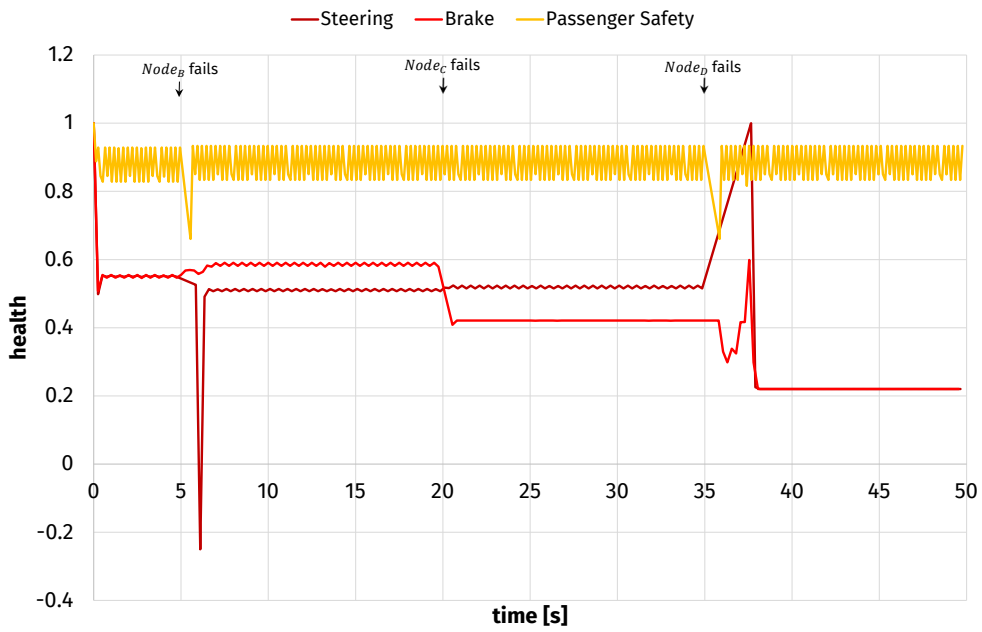


Figure 7.26: Evaluation 5 - Health of the three most important applications in the system.

time [s]	Event	Actions (*Chameleon* instance)	Components	Fitness (before action)	Fitness (after action)	Reward (measured)
5	$Node_B$ fails					
5.5		A1: `RestartDeadApp` (A)	*Passenger Safety*: $Node_D$	0.13605	0.112684	-0.023366
5.8		A2: `RestartDeadApp` (A)	*Steering*: $Node_D$	0.112685	0.57859	0.465905
6.21		A3: `TuneLeastImportantApp` (D)	*Stability and Drive Dynamics*: Factor 2	0.415416	0.313172	-0.102244
8.22		A4: `RestartDeadApp` (A)	*Driver Warning*: $Node_C$	0.313172	0.665961	0.352789
10.12		A5: `RestartDeadApp` (A)	*Lights*: $Node_C$	0.182811	0.658373	0.475562
20	$Node_C$ fails					
20.52		A6: `RestartDeadApp` (A)	*Brake*: $Node_A$	0.084679	0.517752	0.433073
23.32		A7: `RestartDeadApp` (A)	*Driver Warning*: $Node_D$	0.213203	0.56628	0.353077
25.22		A8: `RestartDeadApp` (A)	*Lights*: $Node_D$	0.089786	0.56599	0.476204
35	$Node_D$ fails					
35.52		A9: `RestartDeadApp` (A)	*Passenger Safety*: $Node_D$	0.023876	-0.017908	-0.041784
35.82		A10: `RestartDeadApp` (A)	*Passenger Safety*: $Node_A$	-0.017908	-0.453675	-0.435767
36.12		A11: `PauseLeastImportantAppFast` (A)	*Driver Warning*	-0.453675	-0.491015	-0.03734
36.42		A12: `PauseLeastImportantAppFast` (A)	*Data Repository*	-0.491014	-0.337163	0.153851
36.72		A13: `PauseLeastImportantAppFast` (A)	*Stability and Drive Dynamics*	-0.337164	-0.218061	0.119103
37.02		A14: `PauseLeastImportantAppFast` (A)	*Lights*	-0.218061	-0.293416	-0.075355
37.32		A15: `PauseLeastImportantAppFast` (A)	*Powertrain*	-0.293416	-0.339557	-0.046141
37.62		A16: `RestartDeadApp` (A)	*Steering*: $Node_A$	-0.339558	0.14897	0.488528

Table 7.7: Evaluation 5 - Events and adaptation actions.

7.3.8. Evaluation 6 - Handling Failures and Extreme Overload Situations with Application Specific Rules

Now it will be evaluated how application specific rules can speed up the recovery of specific extremely high important applications. Therefore, application specific emergency rules are added to the basic rule set. They strictly distinguish between the most important applications *Brake*, *Steering* and *Passenger Safety* with the *importance* level 7 and all other applications with an *importance* level below 7. There are three new rules (where the first two extent the basic rule set while the third replaces the `If RestartMightBeUseful Then RestartDeadApp` rule from the basic rule set) together with three new sets and actions, which can be found in Appendix C.

1. Simultaneous pause of applications below *importance* level 7:
   - The first new rule detects if one or more *importance* level 7 applications have failed.
   - In response, it quickly creates space by pausing all applications with a *importance* level below 7 simultaneously using a single action.
   - This action efficiently frees up resources and prepares for the subsequent restart of the failed *importance* level 7 applications.
2. Quick restart of all failed *importance* level 7 applications:
   - The second new rule focuses on promptly restarting all failed *importance* level 7 applications.
   - It initiates a single action to restart these applications as quickly as possible.
   - This ensures the restoration of critical functionalities provided by *importance* level 7 applications without significant delay.
3. Sequential restart of paused applications below level 7:
   - After the failed *importance* level 7 applications have been restarted, this rule takes care of restarting the paused applications below *importance* level 7 one by one, beginning with the highest importance.
   - By using this order of restarts, the system ensures that essential functionalities are restored according to their *importance*.

For the evaluation, the same experiment as in the previous section has been conducted. Table 7.8 shows the events and actions of the adaptation mechanism. The above described strategy of the emergency rule set is clearly evident in Figures 7.27 and 7.28. After each node failure, low and medium important applications are paused to make room for the following simultaneous restart of the important applications (actions A1-A2, A8-A9, A13-A14). Afterwards, the paused applications are restarted/unpaused in order of decreasing *importance* until the resources are no longer sufficient for another unpausing (actions A3-A7, A10-A12). As a result of the strategy, there aren't any unhealthy applications in the system (cf. Figure 7.27) because all the lower important

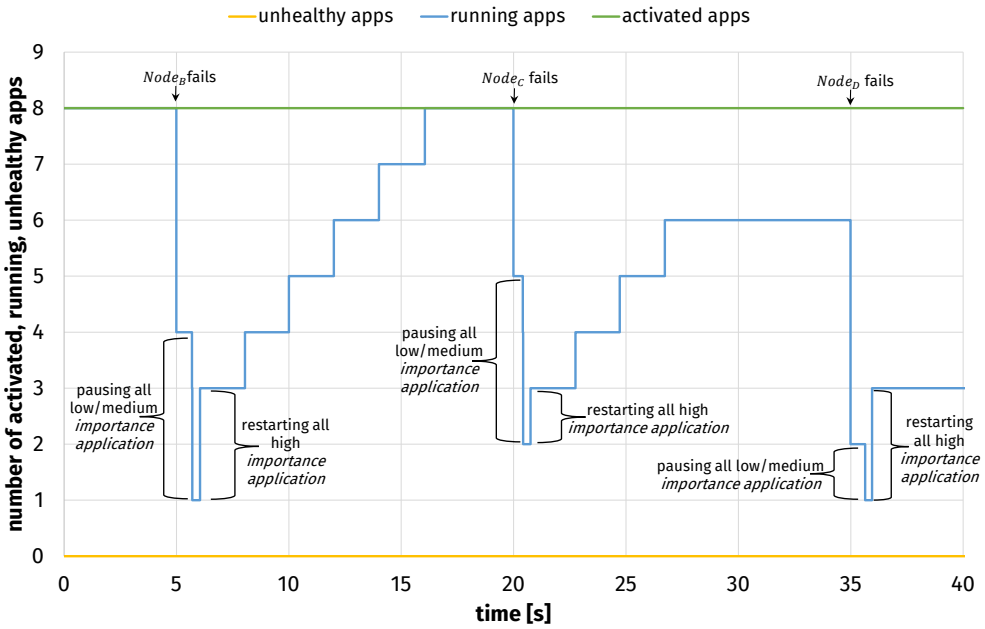


Figure 7.27: Evaluation 6 - Number of unhealthy, running and activated applications.

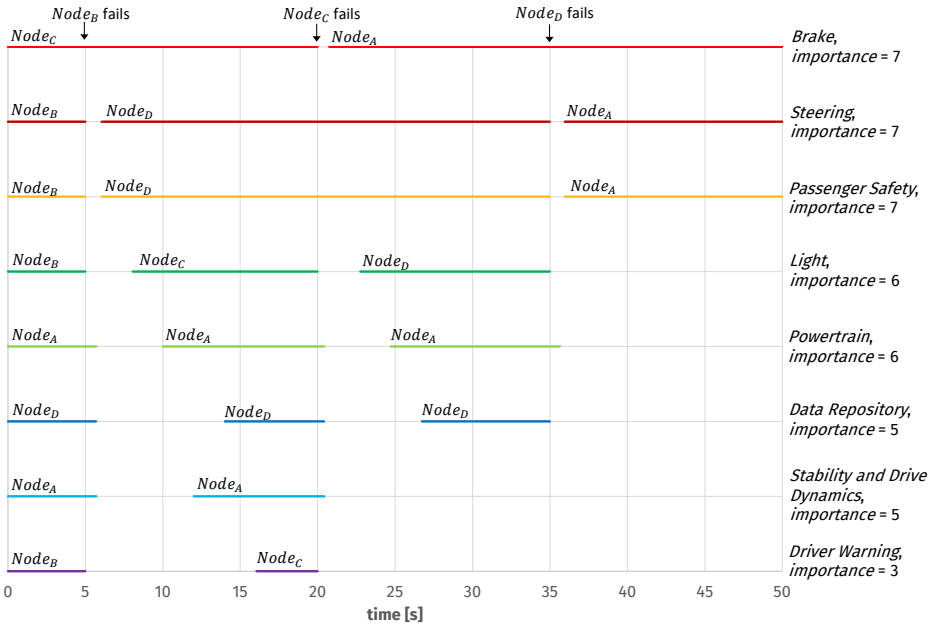


Figure 7.28: Evaluation 6 - Allocation of applications on nodes ordered by *importance*.

applications are paused immediately so the high important applications can use all resources. Then, applications of lower importance are restarted only if there is capacity left. Therefore, the health of the high important applications shown in Figure 7.29 is in a very good state the whole time. In comparison to the previous evaluation, there is no drop of the health value below zero. This good state of the high important applications

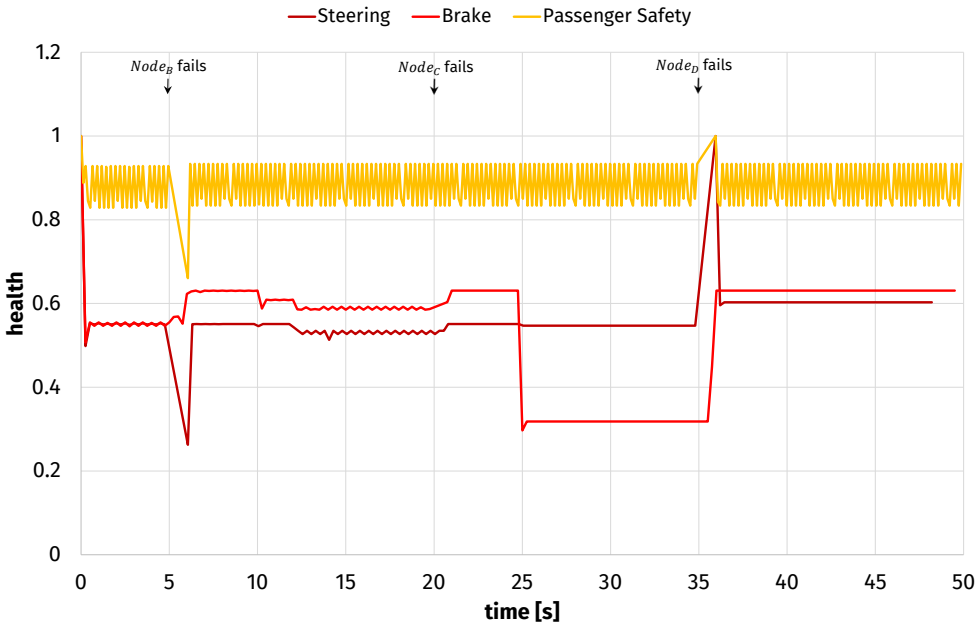


Figure 7.29: Evaluation 6 - Health of the three most important applications in the system.

is due to the usage of the node life sign for the simultaneously detection of application crashes[1] caused by node failure and the strict distinction between the high important application (*Brake*, *Steering* and *Passenger Safety*) and all other application which lead to prioritized and fast restarting of high important applications. For example, the overall adaptation process to restart the high important applications *Steering* and *Passenger Safety* after the failure of $Node_D$ takes only $0.94s$ with the emergency rules, which is nearly 3-times faster as in the previous evaluation. This fast reaction time comes on the expense of the other applications in system (as intended with the prioritization in the emergency rules), which is also reflected in the fitness (cf. Figure 7.30). It drops below zero after each node failure because the fitness reflects not only the number of crashed but also the number of paused applications in the system. Thus, the simultaneous pausing of multiple applications after a failure of a node, together with the crashed application, lead to a drop of the fitness. However, each time the fitness recovers quickly to a positive value due to the successive restart

[1] The simultaneous detection of application crashes by the node life sign avoids *importance* anomalies where lower priority applications are restarted before higher important application because their death has not been detected yet (cf. Sections 7.3.7 and 7.3.6).

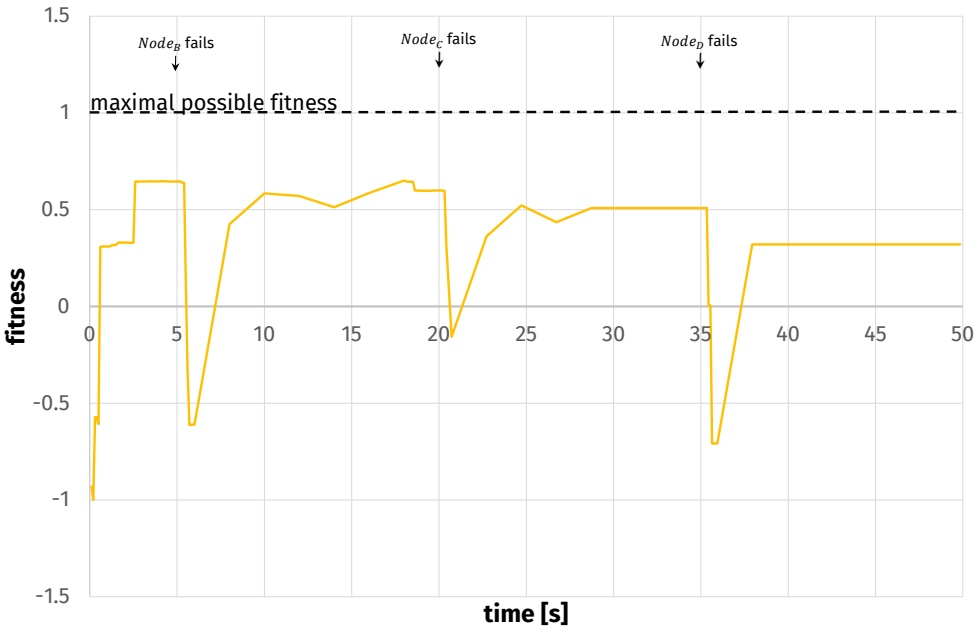


Figure 7.30: Evaluation 6 - System fitness.

of the applications in order of their *importance*[1].

Conclusion The evaluation shows that application specific rules can significantly speed up the recovery process for high important applications at the cost of lower important applications. This demonstrates the flexibility of the rule based adaptation concept and the language *Rango*. Besides the strict threshold between high and lower importance, the speedup is also caused by the capability to start and pause several applications simultaneously by the set concept of *Rango*. Furthermore, the use of the node life sign age instead of the application life sign age offers the advantage to detect applications crashed due to node failures at once.

[1]The importance is also taken into account in the fitness calculation (cf. Section 5.2.3).

time [s]	**Event**	**Actions (*Chameleon* instance)**	**Components**	**Fitness (before action)**	**Fitness (after action)**	**Reward (measured)**
5	$Node_B$ fails					
5.7		A1: `PauseAllLowMediumImportanceApps` (A)	*Lights*, *Driver Warning*, *Data Repository*, *Stability and Drive Dynamics*, *Powertrain*	-0.36981	-0.610137	-0.240327
6		A2: `RestartAllHighImportanceDeadApps` (A)	*Steering*, *Passenger Safety*: $Node_D$	-0.610136	0.424859	1.034995
8		A3: `RestartLowMediumImportantDeadApp` (A)	*Lights*: $Node_C$	0.424859	0.504174	0.079315
10		A4: `UnpauseMostImportantLocalApp` (A)	*Powertrain*	0.504173	0.533886	0.029713
12		A5: `UnpauseMostImportantLocalApp` (A)	*Stability and Drive Dynamics*	0.533887	0.512797	-0.02109
14.01		A6: `UnpauseMostImportantLocalApp` (D)	*Data Repository*	0.512797	0.586531	0.073734
16.02		A7: `RestartLowMediumImportantDeadApp` (A)	*Driver Warning*: $Node_C$	0.587143	0.651019	0.063876
20	$Node_C$ fails					
20.42		`A8: PauseAllLowMediumImportanceApps` (A)	*Lights*, *Driver Warning*, *Data Repository*, *Stability and Drive Dynamics*, *Powertrain*	0.551913	-0.156417	-0.70833
20.72		`A9: RestartAllHighImportanceDeadApps` (A)	*Brake*: $Node_A$	-0.156416	0.36234	0.518756
22.72		`A10: RestartLoveMediumImportantDeadApp` (A)	*Lights*: $Node_D$	0.36234	0.441468	0.079128
24.72		`A11: UnpauseMostImportantLocalApp` (A)	*Powertrain*	0.441468	0.435052	-0.006416
26.73		`A12: UnpauseMostImportantLocalApp` (D)	*Data Repository*	0.435052	0.508657	0.073605
35	$Node_D$ fails					
35.64		`A13: PauseAllLowMediumImportanceApps` (A)	*Lights*, *Data Repository*, *Powertrain*	-0.506824	-0.707777	-0.200953
35.94		`A14: RestartAllHighImportanceDeadApps` (A)	*Steering*, *Passenger Safety*: $Node_A$	-0.707777	0.320886	1.028663

Table 7.8: Evaluation 6 - Events and adaptation actions.

7.3.9. Evaluation 7 - Handling of Design Flaws

This evaluation gives an example on how design flaws can be handled by appropriate rules. The same application scenario as for the previous evaluations together with the basic rule set is used. Thereby, all eleven applications are activated simultaneously at system startup. No events are triggered during runtime, but an initial design flaw is set: No priorities are assigned to the applications and priority based scheduling is deactivated for the computation nodes ($Node_A$, $Node_B$, $Node_C$ and $Node_D$) and the communication channel ($Comm_1$). Table 7.9 shows the actions of the adaptation mechanism.

time [s]	Actions (*Chameleon* instance)	Components	Fitness (before action)	Fitness (after action)	Reward (measured)
0.2	A1: `ActivatePriorityNodeScheduling` (A)	$Node_A$	-2.484305	-0.251194	2.233111
	A2: `ActivatePriorityNodeScheduling` (B)	$Node_B$	-1.855823	0.375527	2.23135
2.21	A3: `TuneLeastImportantApp` (A)	*Infotainment*: Factor 10	-0.251194	-0.087314	0.16388
	A4: `TuneLeastImportantApp` (B)	*Infotainment*: Factor 10	0.375527	0.255901	-0.119626
	A5: `ActivatePriorityNodeScheduling` (C)	$Node_C$	-0.251194	-0.087314	0.16388
	A6: `ActivatePriorityNodeScheduling` (D)	$Node_D$	-0.251194	-0.087314	0.16388
4.22	A7: `ActivatePriorityCommScheduliung` (A)	$Comm_1$	-0.087314	0.503002	0.590316
	A8: `ActivateNodePreemption` (B)	$Node_B$	0.255901	0.503002	0.247101
	A9: `ActivateNodePreemption` (C)	$Node_C$	-0.087314	0.503002	0.590316
	A10: `ActivateNodePreemption` (D)	$Node_D$	-0.087314	0.503002	0.590316
6.23	A11: `TuneLeastImportantApp` (A)	*Navigation*: Factor 5	0.503003	0.591159	0.088156
	A12: `ActivateCommPreemtion` (B)	$Comm_1$	0.503003	0.591159	0.088156
	A13: `UntuneMostImportanLocalApp` (D)	*Infotainment*: Factor 9.5	0.503003	0.591159	0.088156
10.24 - 56.24	A14-A37: `UntuneMostImportantLocalApp` (D)	*Infotainment*, *Navigation*: Factor 1	0.591159	0.678232	various

Table 7.9: Evaluation 7 - Adaptation actions.

The lack of the priority based scheduling on the nodes and communication channel at system start is clearly visible in the system fitness and the number of unhealthy applications shown in Figures 7.31 and 7.32.

The fitness increases strongly with the activation of the priority based scheduling[1] on the nodes and communication channel by the corresponding rules in the basic rule set (third group of rules, cf. Section 7.3.2). With the further activation of preemption for several of these components[2], the fitness reaches a very good positive value. The same observation can be made for the number of unhealthy applications (cf. Figure 7.32). After the priority based scheduling and preemption is activated by the *Chameleon* instances, all activated applications are running healthy.

Furthermore, the initial tuning of the applications *Navigation* and *Infotainment* to the allowed limits (actions A3, A4, A11) can be gradually removed (actions A13-A37).

Conclusion This evaluation shows that *Chameleon* is able to correct certain design flaws by appropriate rules within the scope of the system's capabilities.

[1] The priorities are assign to the application autonomously at runtime based on RMS.

[2] By applying the rules, the autonomous adaptation process determines that it is not necessary to activate preemption on all of these components to bring the system to a healthy state. Preemption on $Node_A$ can e.g. be omitted.

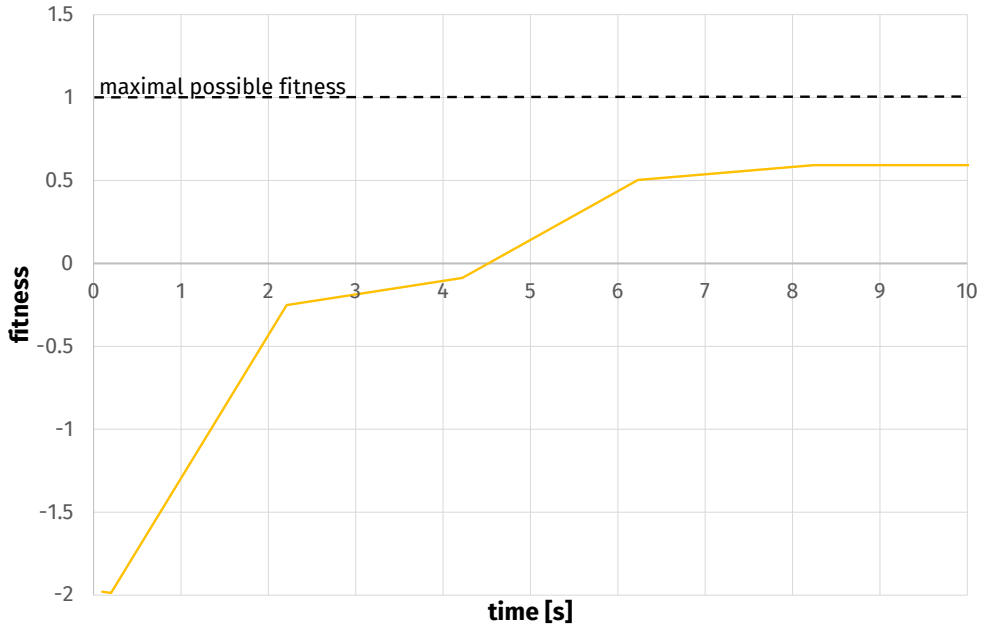


Figure 7.31: Evaluation 7 - System fitness.

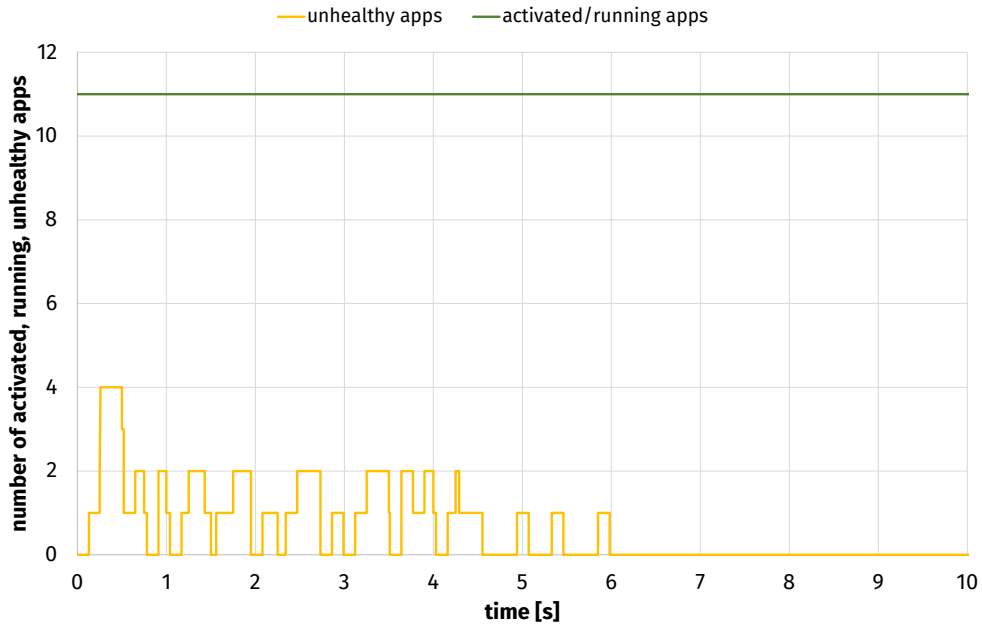


Figure 7.32: Evaluation 7 - Number of unhealthy, activated and running applications.

7.3.10. Evaluation 8 - Effects of Learning

The following evaluation aims to demonstrate the effect of learning by the reward. Therefore, an additional rule is introduced which has the same condition as an already existing rule but a different action. The rule engine can now learn which of both rules performs better under this condition and therefore has to be preferred.

Thus, the new rule

```
If TuningUnhealthyAppsMightBeUseful
Then TuneLocalUnhealthyAppsAlternative Lag 2
Note "Alternative rule if we have unhealthy
tunable apps but comm is ok, lag due to
race conditions"
```

has been added to the already existing rule with the same condition

```
If TuningUnhealthyAppsMightBeUseful
Then TuneLocalUnhealthyApps Lag 2
Note "Rule if we have unhealthy tunable apps
but comm is ok, lag due to race conditions"
```

with the alternate action

```
DefineAction TuneLocalUnhealthyAppsAlternative :
TunePeriodAndPriority LocalUnhealthyTunableApps
To 0.5
Note "tune a local unhealthy app to 0.5"
```

to the already existing action

```
DefineAction TuneLocalUnhealthyApps :
TunePeriodAndPriority LocalUnhealthyTunableApps
To Current Limit
Note "tune a local unhealthy app to its limit"
```

Exactly the same evaluation scenario and events as for evaluation 1 are used. The only difference is the additional rule shown above. Table 7.10 outlines the events and the adaptation actions. It can be seen that the new rule is executed exactly once at time 57.65 (action A6) . The resulting reward is strongly negative (-0.815) which can also be seen in a drop of fitness and the peak of unhealthy applications at that time in Figures 7.33 and 7.34. Afterwards, the new rule is never applied again. Instead, the alternate rule with the same condition is used to bring the system back into a healthy state (actions A7 and A9).

Conclusion This evaluation shows that *Chameleon* is able to learn from past experience. If a rule has not the desired effect, the reward decreases and thus is no longer applied.

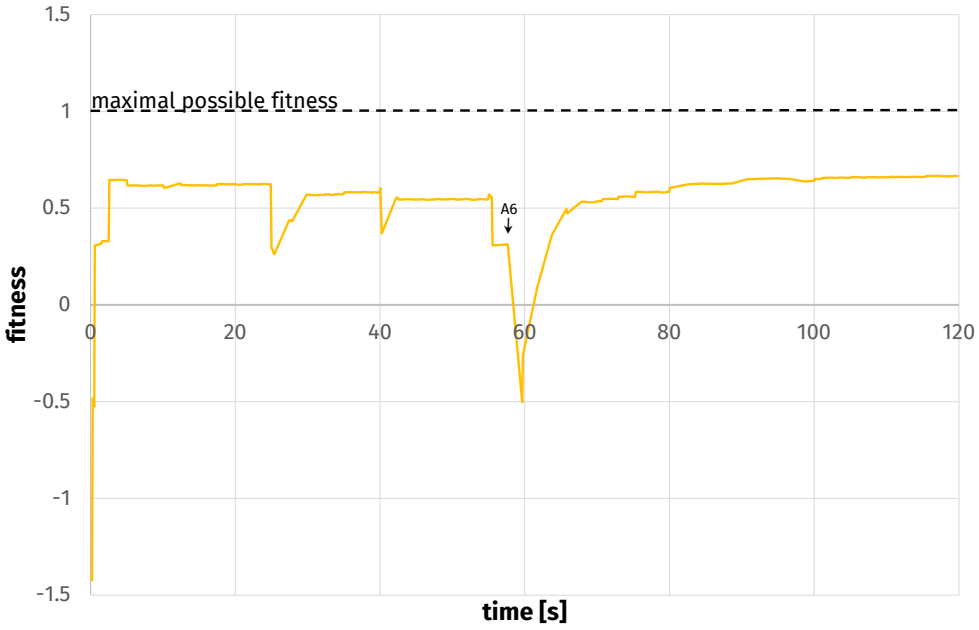


Figure 7.33: Evaluation 8 - System fitness.

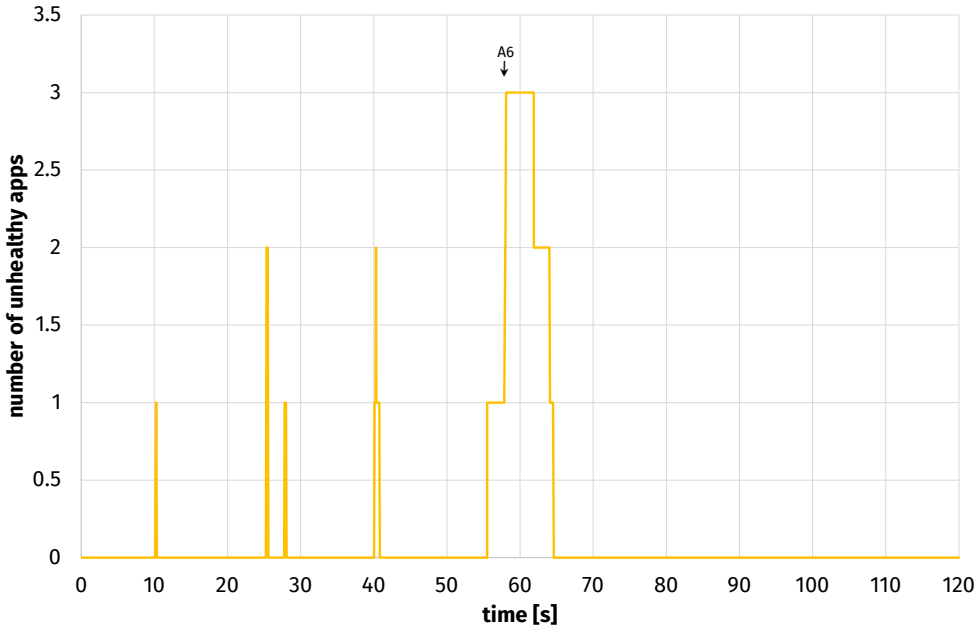


Figure 7.34: Evaluation 8 - Number of unhealthy applications.

time [s]	Event	Actions (*Chameleon* instance)	Components	Fitness (before action)	Fitness (after action)	Reward (measured)
10	Slow down $Node_A$ (10 to 7.5 Mips)					
10.3		A1: `RelocateLocalUnhealtyApp` (A)	*Stability and Drive Dynamics*: $Node_D$	0.347946	0.627541	0.279595
25	Start *CruiseControl* ($Node_C$)					
25.41		A2: `TuneLeastImportantApp` (C)	*Driver Warning*: Factor 2	0.283161	0.436721	0.15356
27.81		A3: `TuneLeastImportantApp` (C)	*Cruise Control*: Factor 2	0.327877	0.569542	0.241665
40	Start *Infotainment* ($Node_D$)					
40.25		A4: `CompressLeastImportantApp` (D)	*Infotainment*: Factor 0.8	0.369106	0.554585	0.185479
55	Start *Navigation* ($Node_D$)					
55.55		A5: `CompressLeastImportantApp` (D)	*Navigation:* Factor 0.8	0.307556	0.312089	0.004533
57.65		A6: `TuneLocalUnhealthyAppsAlternative` (D)	*Navigation:* Factor 0.5	0.313027	-0.502291	-0.815318
59.75		A7: `TuneLocalUnhealthyApps` (D)	*Stability and Drive Dynamics*: Factor 2	-0.502291	0.092478	0.594769
61.75		A8: `RelocateLocalUnhealthyApp` (D)	*Stability and Drive Dynamics*: $Node_A$	0.092478	0.342574	0.250096
63.75		A9: `TuneLocalUnhealthyApps` (D)	*Navigation*: Factor 5	0.342574	0.495224	0.15265
65.85		A10: `TuneLeastImportantApp` (D)	*Infotainment*: Factor 10	0.495224	0.533213	0.037989
70	Speed up $Node_A$ (7.5 to 10 Mips)					
80	Stop *Cruise Control*					
80.65		A11: `UntuneMostImportantLocalApp` (D)	*Navigation*: Factor 4.5	0.608246	0.622654	0.014408
82.65		A12: `UntuneMostImportatnLocalApp` (D)	*Navigation*: Factor 4	0.622654	0.625853	0.003199
84.65		A13: `UntuneMostImportatnLocalApp` (D)	*Navigation*: Factor 3.5	0.625853	0.62337	-0.002483
86.65		A14: `UntuneMostImportatnLocalApp` (D)	*Navigation*: Factor 3	0.623371	0.625141	0.00177
88.65		A15: `UntuneMostImportatnLocalApp` (D)	*Navigation*: Factor 2.5	0.625141	0.645066	0.019925
90	Stop *Infotainment*					
90.65		A16: `UntuneMostImportatnLocalApp` (D)	*Navigation*: Factor 2	0.645066	0.645681	0.000615
92.65		A17: `UntuneMostImportatnLocalApp` (D)	*Navigation*: Factor 1.5	0.645682	0.644233	-0.001449
94.65		A18: `UntuneMostImportatnLocalApp` (D)	*Navigation*: Factor 1	0.644232	0.650314	0.006082
94.66		A19: `UntuneMostImportantLocalApp` (A)	*Stability and Drive Dynamics*: Factor 1.5	0.653509	0.648119	-0.00539
		A20: `UntuneMostImportatnLocalApp` (B)	*Driver Warning*: Factor 1.5	0.654908	0.642723	-0.012185
96.67		A21: `UntuneMostImportantLocalApp` (B)	*Driver Warning*: Factor 1	0.642724	0.637839	-0.004885
100	Stop *Navigation*					

Table 7.10: Evaluation 8 - Events and adaptation actions.

7.3.11. Evaluation 9 - Exploiting the potential of CPN

So far, extensive evaluations of the adaptation mechanisms have been conducted on a single mixed-critical CPS. Now, the benefits of a mixed-critical CPN consisting of more than one mixed-critical CPS will be investigated. Therefore, two vehicles (V_1, V_2) of the application scenario used before are combined as depicted in Figure 7.35.

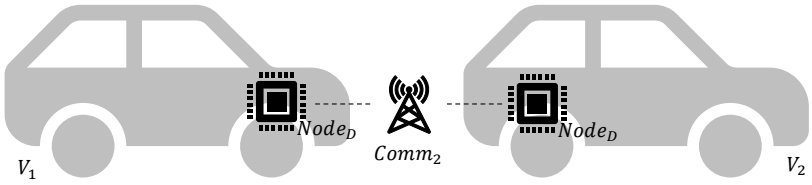


Figure 7.35: Evaluatin 9 - Connection of two vehicles V_1 and V_2 via a wireless connection.

Initially, both vehicles are isolated. After 15 seconds, both vehicles are interconnected. This is done via a wireless connection established between the $Node_D$ of both vehicles. After 100 seconds, both vehicles are separated again. The same events as in evaluation 1 are applied to V_1. The only difference is that after 100 seconds the *Navigation* application is not stopped like in evaluation 1, but kept active after the separation of the two vehicles. During the whole evaluation, V_2 is left in its initial state. Table 7.11 shows these events and the adaptation actions performed in both vehicles. Before the vehicles are connected, the performed adaptation action (A1) of evaluation 1 and evaluation 9 is identical. Afterwards, vehicle V_1 uses local adaptation actions first. This results in almost the same actions as in evaluation 1 (actions A2-A6). After 59.53 seconds, a *Chameleon* instance in V_1 exploits the potential of the additional resources due to the connection to V_2 (CPN) and relocates the *Navigation* application to a node of vehicle V_2 (action A7). Figure 7.36 shows the comparison of the period of the *Navigation* application in evaluation 1 (CPS) and evaluation 9 (CPN). In evaluation 1, the *Navigation* is tuned to a factor of 5 after the activation until the system load is decreased (speed up of $Node_A$, stop of applications *Cruise Control* and *Infotainment*). In evaluation 9, the *Navigation* application is also shortly tuned to a factor of 5 after the activation and relocation, but then this factor is quickly decreased step by step down-to 1.5. Thus, less tunning is needed due to the exploitation of the additional resources of the CPN. The overall better system condition is also visible in the system fitness illustrated in Figure 7.37. Of course, the fitness values of both evaluations converge again with the reduction of the load in the entire system.

After the disconnection of vehicles V_1 and V_2, the *Navigation* application of V_1 is stopped as an orphan in V_2 (action A19) and restarted in V_1 (action A20) after 1.57 seconds. This time also complies to the upper bound given by Equation 7.12 ($t_{ReactionTimeCrash_k} \leq l_{LifeSignLimit} \cdot t_{MonitoringPeriod_{remote_k}} + t_{Communication_k} + t_{\texttt{EvaluationPeriod}}$). With a remote monitoring period of $1000ms$ for the *Navigation*

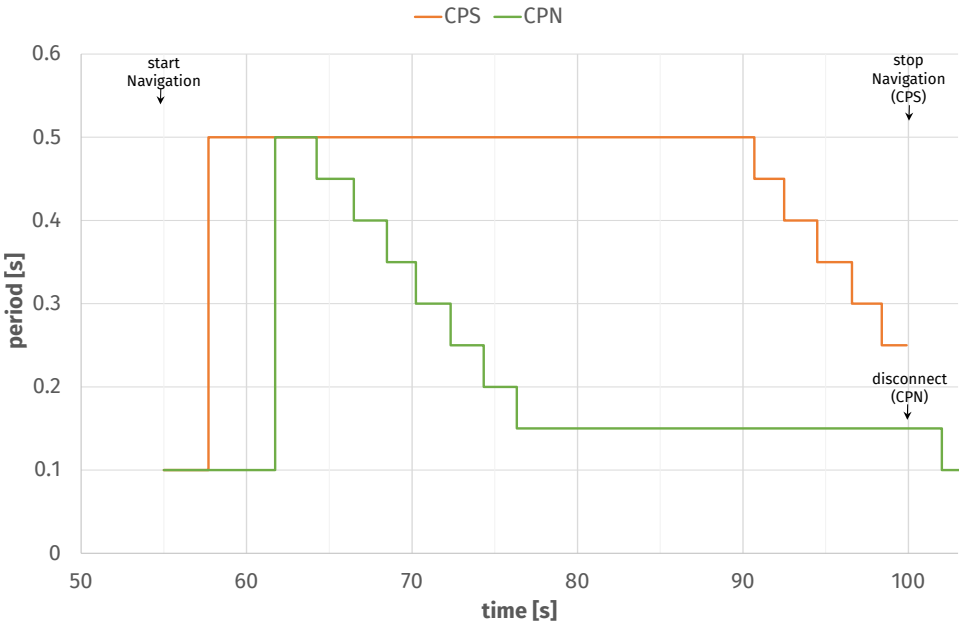


Figure 7.36: Evaluatin 9 - Comparison of the period of the *Navigation* application in evaluation 1 (CPS) and evaluation 9 (CPN) while *Navigation* is running in both evaluations.

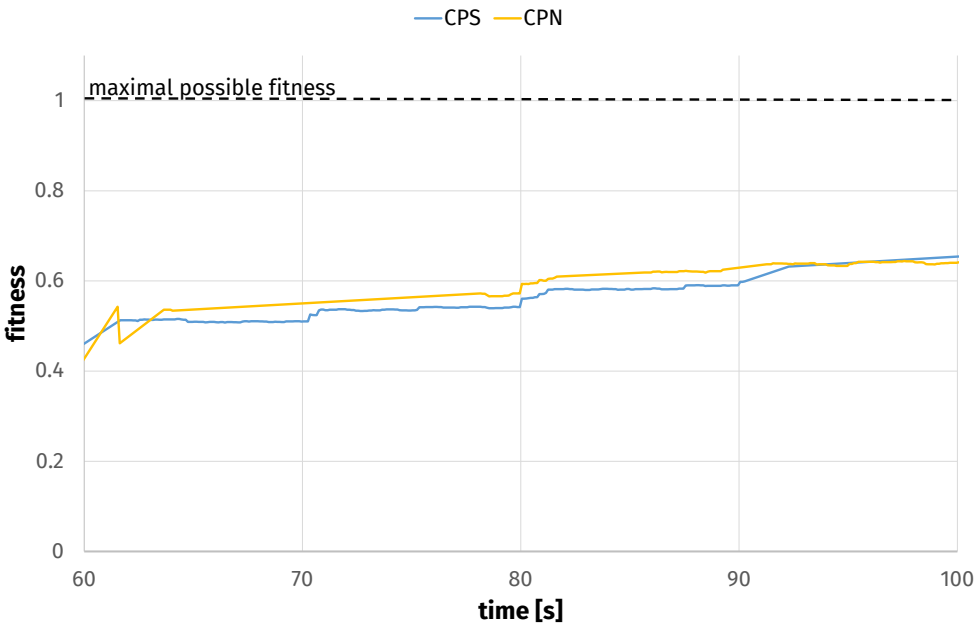


Figure 7.37: Evaluation 9 - Comparison of the system fitness of evaluation 1 (CPS) and evaluation 9 (CPN) while *Navigation* is running in both evaluations.

application[1], a life sign limit of 2 times the remote monitoring period, an evaluation period of $100ms$ and a communication time of $50ms$, the upper bound for restarting the *Navigation* locally after the connection loss calculates to 2.15 seconds.

Conclusion The adaptation mechanism can use the resources of CPNs and adapt the system through global and local adaptation actions. The exploitation of abilities and resources of the whole CPN leads to more adaptation options and better results.

[1]The *Navigation* application period is $100ms$, cf. Table 7.2 $\Rightarrow t_{MonitoringPeriod_{local_{Navigation}}} = 5 \cdot 100ms \Rightarrow t_{MonitoringPeriod_{remote_{Navigation}}} = 2 \cdot 5 \cdot 100ms$.

time [s]	Event	Actions (*Chameleon* instance)	Components	Fitness (before action)	Fitness (after action)	Reward (measured)
10	V_1: Slow down $Node_A$					
10.30		A1: `RelocateLocalUnhealthyApp` (V_1:A)	*Stability and Drive Dynamics* (V_1): NodeD (V_1)	0.350277	0.630047	0.27977
15.00	Connect V_1 and V_2					
25	V_1: Start *Cruise Control* ($Node_C$)					
25.41		A2: `TuneLeastImportantApp` (V_1:C)	*Driver Warning* (V_1): Factor 2	0.412277	0.443544	0.031267
27.51		A3: `TuneLeastImportantApp` (V_1:C)	*Cruise Control* (V_1): Factor 2	0.21784	0.574062	0.356222
40	V_1: Start *Infotainment* ($Node_D$)					
40.23		A4: `CompressLeastImportantApp` (V_1:D)	*Infotainment* (V_1): Factor 0.8	0.464409	0.560813	0.096404
55	V_1: Start *Navigation* ($Node_D$)					
55.53		A5: `CompressLeastImportantApp` (V_1:D)	*Navigation* (V_1): Factor 0.8	0.318298	0.320421	0.002123
57.53		A6: `TuneLeastImportantApp` (V_1:D)	*Infotainment* (V_1): Factor 10	0.320422	0.392301	0.071879
59.53		A7: `RelocateLocalUnhealthyApp` (V_1:D)	*Navigation* (V_1): NodeD(V_2)	0.3923	0.54285	0.15055
61.64		A8: `TuneLocalUnhealthyApps` (V_2:D)	*Navigation* (V_1): Factor 5	0.609599	0.614385	0.004786
64.14 - 68.14		A9-A11: `UntuneMostImportantLocalApp` (V_2:D)	*Navigation* (V_1): Factor 3.5	0.612619	0.614034	various
70	V_1: Speed up $Node_A$					
70.14 - 76.14		A12-A15: `UntuneMostImportantLocalApp` (V_2:D)	*Navigation* (V_1): Factor 1.5	0.614033	0.610731	various
80	V_1: Stop *Cruise Control*					
81.65		A16: `UntuneMostImportantLocalApp` (V_1:B)	*Driver Warning* (V_1): Factor 1.5	0.623243	0.631515	0.008272
83.65		A17: `UntuneMostImportantLocalApp` (V_1:B)	*Driver Warning* (V_1): Factor 1	0.631515	0.629142	-0.002373
89.26		A18: `UntuneMostImportantLocalApp` (V_1:D)	*Infotainment* (V_1): Factor 9.5	0.624809	0.63713	0.012321
90	V_1Stop *Infotainment*					
100	Disconnect V_1 and V_2					
100.47		A19: `KillForeignOrphans` (V_2:D)	*Navigation* (V_1)	0.61109	0.617914	0.006824
101.57		A20: `RestartDeadApp` (V_1:A)	*Navigation* (V_1): NodeA(V_1)	0.337699	0.664436	0.326737
101.87		A21: `UntuneMostImportantLocalApp` (V_1:A)	*Navigation* (V_1): Factor 1	0.664436	0.518178	-0.146258
103.97		A22: `TuneLocalUnhealtyApps` (V_1:A)	*Navigation* (V_1): Factor 5	0.518177	0.556942	0.038765
106.07		A23: `TuneLeastImportantApp` (V_1:A)	*Stability and Drive Dynamics* (V_1): Factor 2	0.556942	0.614247	0.057305
108.07 - 122.07		A24-A31: `UntuneMostImportantLocalApp` (V_1:A)	*Navigation* (V_1): Factor 1	0.614248	0.624628	various
124.08 - 126.08		A32-33: `UntuneMostImportantLocalApp` (V_1:D)	*Stability and Drive Dynamics* (V_1): Factor 1	0.646111	0.630105	various

Table 7.11: Evaluation 9 - Events and adaptation actions.

8. Related Work

Research in MC-CPNs covers a wide range of different topics, which has been addressed by a large number of research groups in academia as well as in industries in the last years. Depending on the goals of the various approaches, a combination of generic research fields is covered. *Chameleon* mainly contributes to the fields of middlewares, self-organization, mixed-criticality and learning adaptation. Therefore, related work on these topics is presented in the following.

8.1. Middleware

Middleware is a generic concept to ease the development of distributed systems. It relieves the developer of taking care for the physical distribution of an application. A middleware is responsible to identify, locate and access the distributed entities in the system. It also handles events and errors occurring in distributed operations. Furthermore, heterogeneous environments regarding operating systems, computation nodes and communication links are concealed from the application developer.

Due to varying application fields and research objectives, the concepts for middleware can be divided into different categories. Here, relevant categories for this thesis are addressed.

8.1.1. Classical Middlewares

Classical middleware approaches mainly focus on the aspects of hiding the physical distribution and heterogeneity. Depending on the granularity and access methods of the distributed entities, service-based, object-based, procedure-based or message-based middleware architectures can be distinguished. Service-base approaches allow the distribution and interaction on the level of entire services, which can be invoked by clients or other services. Object-based concept use objects known from the object oriented programming paradigm as distributed entities. In procedure-based approaches, procedures resulting from the paradigm of structured programming form the granularity of distribution. Message-based middlewares focus on the pure message exchange between any distributed item. Therefore, they can be used as a basis for the other concepts.

Due to the popularity of the object oriented programming paradigm, object-based middleware concepts has been the major role model for classical middlewares. A good example for this is CORBA (Common Object Request Broker Architecture) [Vin97], which forms a standard for object-oriented middlewares developed in cooperation of acadamia and industries. It allows to map interface specifications to different programming languages and the interoperable inter-orb protocol (IIOP) provides

interaction across different platforms with respect to programming languages and operation system [Vin97]. Therefore, CORBA in combinition with IIOP can be used for the development and deployment of applications in distributed heterogeneous environments but there is no interaction to the physical world provided.

Other such approaches are DCOM [PS98] and .NET [Pro02], both mainly driven by Microsoft. Therefore, they are deeply integrated in the Windows operating system and enable distributed applications for that operating system using different programming languages and hardware platforms.

Another classical middleware not focused on a specific operating system, but on a specific programming language is Java RMI [Onl]. Like for all the other mentioned classical middleware approaches, the interaction to the physical world is not in the focus here.

8.1.2. Cyber-Physical Systems Middlewares

The connection of the physical world and computation devices regards many challenges for systems, especially real-time communication, context-awareness, adaptiveness, heterogeneity of devices as well as applications (criticality) and communication channels, scalability, or security [GLV13; Gun+14; JLS09]. As a result, the landscape of cyber-physical middlewares is diversified. Some of the existing middlewares are presented in the following. Each focuses on different aspects: timining, data distribution, modularity/composition, and heterogeneity.

Zhang, Gill, and Lu present an extension of the TAO middleware for timing aspects [ZGL08]. There, each functionality is seen as a sequence of events and scheduled to devices by a central scheduler. This is different to the approach presented here, where scheduling is dezentralized. Also, the authors do not further stress the communication functionality nor do they consider adaptation and different criticality levels.

RDDS offers a publish/subscribe based data distribution middleware supporting semantic aware communication [KKS12]. Further, this approach enables reactive and proactive adaptations to keep QoS-bounds. Nevertheless, it is focused on data dissemination rather than communication itself. Furthermore, mixed-critical applications are not in the focus of this work.

iLAND enables time-bound reconfigurations of service-oriented soft real-time systems [GLV13]. It offers reconfiguration graphs and paths for service compositions and discovery but neglects to cover specifics of the communication channels for enabling QoS-bounded communication. As well, only soft real-time applications are considered and the mixture of applications with different criticality on same platform is not supported.

[GB15] provides an initial-architecture for a CPS middleware. Thereby, the target is on subsystems where timing deadlines are not hard nor safety critical. Thus, this architecture is not suitable for the goals of this thesis.

In [NCA13], the authors extend the Flexible Time-Triggered (FTT) paradigm by a middleware architecture called FTT-MA. FTT-MA focuses on scheduling and resource management for dynamic reconfiguration at runtime. A very first prototype was

implemented in CORBA. Different to our event-triggered approach, FTT-MA is based on a time triggered protocol. Also, the communication network parameters are not monitored nor adapted at run-time.

None of the presented works follows a comprehensive adaptive approach like *Chameleon*, where the entire system (mixed-critical real-time applications, network nodes, network, sensors and actuators) is addressed (cf. Section 3). Further, networks of CPSs (CPNs) are not regarded by these works..

8.1.3. Self-Organizing Middlewares

As postulated by research hypothesis 2 (c.f. Section 1.2), the middleware is a well suitable place to encapsulate autonomous behavior. This directly leads to the research field of self-organizing middlewares (e.g. [Rot+11; RBP11; NB08; Rat+17]). They enrich the field of middlewares by using different mostly decentralized adaptation mechanisms to provide a set of self-X properties (cf. Section 2.3.1).

Since self-organization itself is an important and vivid research field, this will be discussed more detailed in the next section. First, the basic principles of self-organization will be addressed and then the resulting impact on middleware concepts will be regarded.

8.2. Self-Organization

Self-organization has been a research focus for many years. Publications like [Jet89] or [Whi95] deal with general mechanisms of self-organizing systems, like e.g. emergent behavior, reproduction, etc. These can be used in a wide range of application fields as ecomonics, enterprises or societies.

In the last two decades, self-organization has also be adopted by computer science. The major reason for this is the enormously rising complexity of computer systems in this time frame. It has triggered a considerable number of projects and initiatives. In the following, research efforts on the basic principles of self-organization in computer science will be listed. Then, projects dealing with the encapsulation of self-organization in middleware will be addressed.

8.2.1. Basic Principles

There are many definitions of the term self-organization. According to [Mue+07], self-organizing systems are adaptive systems, which are self-managing and adapt their structure according to the user's expectation and environmental conditions. The control of the self-managing process as well as the structural adaptation are decentralized.

IBM's and DARPAS's Autonomic Computing project [KC03] deals with self-organization of IT servers in networks. The project seeks to adapt and optimize the system behavior in response to changing conditions and requirements autonomously without human intervention. They are characterized by several so-called self-X properties like self-optimization, self-configuration, self-protection and self-healing. The MAPE feedback

loop consisting of four main stages or components: (i) Monitor, (ii) Analyze, (iii) Plan and (iv) Execute was defined to realize these properties. It is an iterative process, which is continuously executed in the background and in parallel to normal server activities similar to the autonomic nervous system. The loop is often augmented with additional component for the knowledge management(cf. Section 2.3.3).

The German Organic Computing Initiative was founded in 2003. Its basic aim is to improve the controllability of complex embedded systems by using principles found in organic entities [VDE03; Sch05]. Organization principles which are successful in biology are adapted to embedded computing systems and used to acquire self-X properties (cf. Section 2.3.1). One of the basic concepts of Organic Computing systems is the Observer-Controller Architecture [Hel+04] which has some similarities to the MAPE-K loop (cf. Section 2.3.1). Here, the system under observation and control is monitored by an observer and guided by a controller based on the observer results. Learning strategies like e.g. LCS are frequently used for the observer. Related work regarding LCS is addressed in Section 8.4.

On the European level, the EU program Future Emerging Technologies FET - complex systems [EU09] also hosted research on self-organization for computing systems. In Australia, the Centre for Complex Systems (CSS) in the Commonwealth Scientific and Industrial Research Organisation (CSIRO) [CSI09] conducted basic research on self-organisation.

Self-organization for embedded systems has been addressed especially at the ESOS workshop [Bri13]. Furthermore, there are several projects related to this topic like ASOC [Lip+05; BBR09], CARSoC [Klu+06; Klu+08] or DoDOrg [JBe+06]. ASOC (Autonomic System on Chip) brings Autonomic Computing to the system on chip level. Such an autonomic system on chip consists of functional elements, which perform the real work. Associated with each functional element is an autonomic element, which monitors and controls its operation. CARSoC (Connective Autonomic Real-time System on Chip) also addressed systems on chip. Here, an network of system on chips is controlled locally and globally by autonomic feedback loops. DodOrg (Digital on Demand Computing Organism) arose from the Organic Computing Initiative. It uses a mammal heart as a blueprint for a reliable and robust computer architecture by mimicking cells and hormones (see also AHS in next section). However, all these projects do not deal with CPS.

A self-healing framework for building resilient CPS which achieves self-healing through structural adaptation is presented in [Rat+17]. In contrast, *Chameleon* is not limited to structural adaptation. Instead also parametric adaptation to modify the behavior of the system is applied.

8.2.2. Encapsulating Self-Organziation into Middleware

Self-Organization is often encapsulated into a middleware. Such a middleware is usually called a self-organizing middleware. For example, several project related to Organic Computing use this approach. In the frame of the DoDOrg project, the Artifical Hormone System (AHS) was introduced [JBe+06; BPR08; RBP11]. The AHS has been

later extended by the Artifical DNA (ADNA) [Bri17].

Another hormone based middleware approach called OCμ respectively AMUN has been proposed in [Rot+11; TTU06]. [Wei+09] describes self-organization in automotive embedded systems.

The CARISMA middleware [NB08] uses an agent based approach featuring the ContractNet protocol, which is a communication protocoll used for auction based negotiation in multi-agent systems [Smi80]. An introduction to multi-agent systems can be found in [Woo09]

All these middleware architectures mainly deal with self-organizing task allocation. In *Chameleon*, task allocation is one adaptation possibility in a wider range of other self-organizing adaptation capabilities. Further, the mentioned self-organizing middlewares are not designed for CPS/CPN and neglect CPS/CPN aspects like sensor/actuator interaction.

Also, in the research area of Wireless Sensor Networks (WSN), task allocation has also been researched for several years using self-organization approaches [Guo+15; Yan+14; Yin+17]. In comparison to *Chameleon*, the focus in the field of WSNs lies on energy-efficiency rather than on safety-related timing constraints.

Besides the online task allocation methods described above, also offline task allocation has been researched. There, the task mapping is solved at design time of the system. Examples for such offline mapping are e.g [Orl+19; Osz+18; Kov+20; Kov+17; Bar04]. However, for *Chameleon* a static offline task mapping is not suitable due to the necessity to adapt at run-time to varying system and operational conditions. The shifting of decisions from the design time to the runtime is one of the classification features of adaptive systems [TSM17; MT17].

For task allocation in mixed-critical systems, the criticality of the tasks has to be respected. This aspect is addressed in the next section.

8.3. Handling Mixed-Criticality

An approach for a self-organizing and dynamic task allocation with prioritization is presented in [HB21]. This is an extension of the bio inspired middleware AHS which is already mentioned above. Assignment priorities similar to the *importance* value to express criticality in *Chameleon* are introduced to determine the order of task assignment. The assignment priorities for tasks also enables self-healing in overload situation by gradually dropping low-priority tasks. Thus, the task allocation on nodes with regard to a criticality expressed by priorities is possible. In comparison to *Chameleon*, the system view is limited to the nodes and the adaptation actions to handle overload situations are limited to task dropping.

There are several other approaches dealing with the handling of mixed-criticality. Those are in general based on different scheduling strategies to ensure the sound execution of critical system tasks:

Hu *et al.* introduces an adaptive real-time scheduling algorithm to handle dynamic multiple-criticality applications based on a least-laxity first strategy. Thereby, a mode-

switch scheme and virtual deadlines to meet the different requirements of multiple-criticality applications are used [HCZ19].

ASDYS is a a dynamic scheduling approach using active strategies where the mixed-criticality is actively treated throughout the scheduling process. The aim of the approach is to minimize the deadline miss rate ratio [Bai+21].

GoodSpread is a framework that statically allocates multi QoS resources to a set of control applications so that each of them meets its performance requirements in all scenarios while using the minimum amount of high-QoS resources [Roy+20]. All of those approaches address mixed-criticality in terms of scheduling only.

Chameleon uses scheduling as one of many adaptation possibilities. The comprehensive approach in combination with the rule language *Rango* allows to consider the criticality in all possible adaptation conditions and actions by the use of the *importance* parameter.

8.4. Learning - LCS and Adaptation Languages

In an insightful overview [DAn19], D'Angelo *et al.* compare which learning techniques are applied in adaptive systems. Approaches that rely on Reinforcement Learning (RL) are dominantly applied to realize simple learning tasks (e.g., with Q-Learning) as well as sophisticated ones (e.g., with LCS). Further, Multi-agent RL (MARL) is often employed to solve problems in a distributed manner when centralized control becomes infeasible [MDC17].

In *Chameleon*, the focus lies on LCS because those and variants — such as the *Extended Classifier System* (XCS) [ST21] by Wilson [Wil95] or *XCS for real-valued input spaces* (XCSR) [WS21] — are well-suitable for real-time systems, have a low computational complexity and have been widely used for implementing adaptive behavior with runtime learning capabilities in various domains. For instance, they have been applied in typical CPS use cases such as self-adaptive traffic management [Ste+17a; STH16], autonomous parameter adjustment of data communication protocols [TH11], or Industry 4.0 [HPH20]. Beyond the domain of CPS, Rosenbauer *et al.* [Ros+21] applied *XCS for function approximation* (XCFS) for automated test case prioritization and Stein *et al.* targeted a smart cameras application [Ste+17b].

In contrast to most LCS based solutions which use simple bitstrings to define rules, *Chameleon* uses *Rango* as a high-level adaptation language to express the rules. To the best of the author's knowledge, none of the approaches mentioned above focuses on supporting the system administrator in the process of the rule set generation. Still, the applied learning principles can be combined with the *Chameleon* approach to learn new rules or optimize existing ones.

In general, several requirement modeling and adaptation languages as well as modelling approaches such as *FLAGS* [BPS10], *CARE* [QJP12], *RELAX* [Whi+10], *LOREM* [BCZ05] or [Ram+12] exist. The main purpose of such languages is the definition of requirements at design time. At runtime, the resulting models can be used as a knowledge base to support the reasoning for adaptation.

Similarly, languages such as *Stitch* [CG12] and *Ctrl-F* [ARS15] focus on the definition of adaptation behavior. The structural specification language *Clafer* [Bak+16] is used to model adaptation behavior and *UML* to model the target system in the *REACT* framework [Pfa20b] and its extension *REACT-ION* [Pfa20a].

[Kru18] focuses on a clear separation of the adaptation decision logic and the rules. Here, an initial rule set can be defined in a spreadsheet.

In comparison to these approaches for specifying adaptation behavior, *Rango* has two key strengths: First, *Rango* is specifically designed for the usage in CPS. It includes, for instance, a large variety of CPS-specific keywords. Second, *Rango* is well-integrated with a learning mechanism while the languages mentioned above do not integrate mechanisms to learn or adjust the models.

Instead of learning rule and language based adaptation, another research stream focuses on the reasoning of adaptation under uncertainty [EKM11; Mor+16; Mor+18; GPB18; Kin+18], which is typical for CPS. Those works do focus on applying statistical approaches such as Markov decision processes or Bayesian optimization to reflect uncertainty. Also, they do not integrate learning. For example, *SimCA** [SWM19] provides a control-theoretic approach that offers guarantees for uncertainty related to system parameters, component interactions, system requirements, and environmental uncertainty.

8.5. Contribution

In summary, *Chameleon* contributes to the state of the art by introducing and combining the following concepts:

- A comprehensive self-adaption mechanism on all levels of the system model is provided.
- This mechanism allows a flexible combination of parametric and structural adaptation actions (relocation, scheduling, tuning, ...) to modify the behavior of the system.
- Real-time constraints of mixed-critical applications (hard real-time, soft real-time, best-effort) are considered in all possible adaptation conditions and actions by the use of the *importance* parameter.
- CPNs are supported by the introduction of different scopes (local, system, global) for the adaptation conditions and actions. This also enables the combination of different scopes for conditions and actions.
- Instantiating the MAPE-K loop by a distributed LCS allows for real-time capable reasoning of adaptation actions which also works on resource-spare systems.
- *Rango* offers an intuitive way to specify an initial rule set for LCS in the context of CPS/CPNs and supports the system administrators in the process of rule set generation.

9. Summary

The thesis presented here deals with the question on how to conveniently, effectively and efficiently handle the management and complexity of MC-CPN's. This question arises due to the increasing complexity of MC-CPN's which can't be handled effectively and efficiently anymore by the system developer without the assistance of the system itself. Thus, efficient, reliable and scalable solutions for managing MC-CPNs and their complexity while ensuring safety are needed.

Based on five research hypotheses (cf. Section 1.2) *Chameleon* — an adaptive middleware for MC-CPN — was designed, developed and implemented in C++. This middleware encapsulates a rule based MAPE-K loop to realize autonomous behavior through the realization of self-X properties. The mixed-criticality is modeled by the introduction of an *importance* parameter which reflects the criticality level of various system tasks. Furthermore, a distributed and comprehensive approach in which the entire system (cf. Section 3) is modeled and subject for management and adaptation, is used. A summary of the contributions of *Chameleon* to the state-of-the art can be found in the previous Section 8.5.

To evaluate the approach, an extensive evaluation was conducted. First the usefulness, usability and performance of the rule language *Rango* has been evaluated in Section 7.1. Afterwards, the real-time behavior of *Chameleon* has been examined in Section 7.2. Finally, the usability of the rule-based MAPE-K adaptation mechanism for managing MC-CPN's and their complexity while considering the mixed-criticality has been investigated in Section 7.3 by an automotive application scenario. Therefore, *Chameleon* has been embedded in a simulated environment (cf. Section 6) and nine evaluation scenarios were conducted which focuses on the following aspects:

- Dynamic load changes and handling of mixed-criticality in overload situations (Evaluation 1, cf. Section 7.3.3).
- Autonomic vs. manual adaptation (Evaluation 2, cf. Section 7.3.4).
- Communication overhead of *Chameleon* (Evaluation 3, cf. Section 7.3.5).
- Handling of failures (Evaluations 4-6, cf. Sections 7.3.6-7.3.8).
- Handling of design flaws (Evaluation 7, cf. Section 7.3.9).
- Effects of learning (Evaluation 8, cf. Section 7.3.10).
- Exploitation of the increased number of available resources in CPNs (Evaluation 9, cf. Section 7.3.11).

9.1. Conclusion

The extensive evaluation presented in Section 7 has demonstrated the validity of the posed five research hypotheses for the design development of *Chameleon* (cf. Section 1.2 and Figure 9.1).

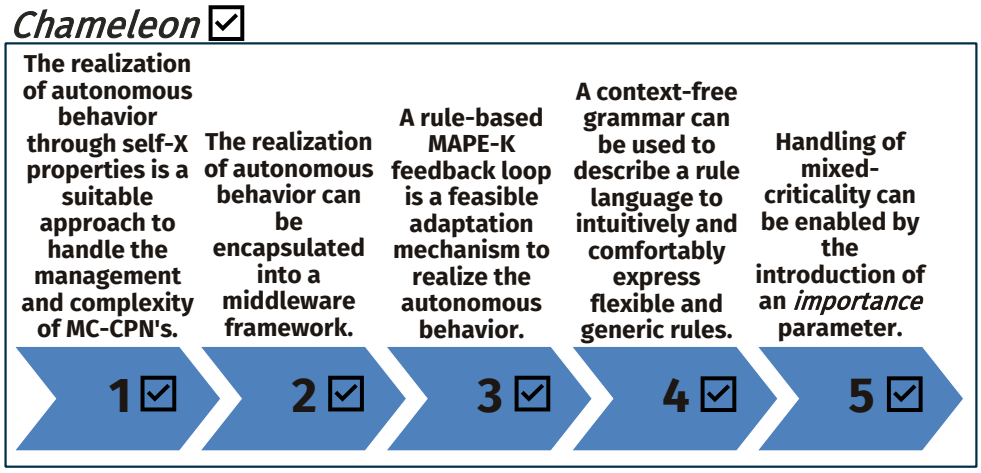


Figure 9.1: Validity of the research hypotheses for the design and development of *Chameleon*.

The realization of autonomous behavior through self-X properties has shown to be a suitable approach to handle the management and complexity of MC-CPN's (*Hypothesis 1*). Furthermore, the encapsulation of this autonomous behavior into a middleware software layer has provided features that allow the MC-CPN to self-organize and adapt to changing conditions and system goals without requiring a central control or human interacting (*Hypothesis 2*). Thereby, the comprehensive approach considers the entire system consisting of applications, computing nodes, communication channels, sensors, actuators and the middleware itself in terms of management and adaptation. Thus, the capabilities of *Chameleon* clearly exceeds specific research in related areas (cf. Section 8) by addressing a multitude of issues simultaneously while effectively reliving the system developers from handling the management and complexity of MC-CPN by themselves.

A rule-based MAPE-K feedback loop has shown to be a feasible adaptation mechanism for realizing the self-X properties for autonomous behavior in the middleware (*Hypothesis 3*). Furthermore, the use of a distributed LCS to instantiate the MAPE-K feedback loop allows for real-time capable reasoning of adaptation actions on resource-spare systems. Finally, *Rango* has turned out to offer an intuitive way to specify an initial rule set for LCS in the context of MC-CPNs and to support the system administrators and developers in the process of rule set generation (*Hypothesis 4*). Due to the distributed nature of MC-CPN, *Chameleon* also uses a distributed rule evaluation (no central control). Therefore, the middleware additionally offers strategies

for interference treatment and reward calculation. The rule language allows for a flexible combination of different scopes for adaptation actions and conditions. Further, it allows to combine various combinations of parametric and structural adaptation actions to modify the behavior of the system while considering real-time constraints of mixed-critical applications. Finally, the introduction of an *importance* parameter in combination with the rule language has demonstrated to be a valid and powerful apporach to handle mixed-criticality (*Hypothesis 5*).

Overall, *Chameleon* enables an autonomous, efficient, effective, reliable and flexible management of MC-CPNs and their complexity while considering the mixed-criticality aspect.

9.2. Outlook

Finally, it's important to consider the next steps and potential avenues for future work and research. The presented thesis has introduced a promising approach for the management of CPNs and their complexity, but there is still much left to be explored and optimized.

To begin with, the learning of *Chameleon* can be improved by establishing strong interference avoidance. As discussed in Section 5.2.3, the reward calculation in a distributed LCS poses the challenge of interferences. In the current implementation of *Chameleon*, these interferences are prevented by introducing and adapting reward delays. Nevertheless, they can occur. In case of an interference is detected, it is handled by reducing the learning rate. Thus, learning can be improved by introducing strong interference avoidance. Several realization options can be investigated and compared in future work. For example, a central interference handler could be implemented. Because this centralized solution would represent a single-point-of failure, also a decentralized solution would be favorable. E.g., a handshake solution with action announcement including their *importance* in the system can be realized. This would also allow to prioritize in case of interference conflicts.

Moreover, the generation of new rules can be explored. In the presented approach, the basic rule set is modified by reward based online learning. This could be extended. Offline learning, e.g. based on genetic algorithms, could be investigated to modify existing rules or to create completely new ones. By mutation, cross-over and selection, condition clauses, action parameters and value ranges of existing rules could be varied. New rules could be created randomly or by genetic combination of well-rewarded existing rules. In an evolutionary process, good rules could prosper while bad ones would become extinct.

Another important step in future work is the real-world deployment and evaluation of *Chameleon* in various application scenarios. In this context, the use of *Chameleon* in wireless sensor networks, which focus more one energy efficiency than mixed-criticality, can also be investigated since the energy aspect can easily be integrated through a modification of the health value derivation.

Additionally, other adaptation mechanisms (e.g. bio-inspired approaches from the

field of Organic Computing or deep learning neural networks) could be investigated in the middleware framework.

In summary, the presented thesis has laid the groundwork for future research, and the proposed modifications and improvements offer potential for advancing the field. By onward exploring and optimizing the presented approach, the performance and effectiveness of the management of complex MC-CPNs can be further enhanced.

A. Contextfree LL(1) - grammar – *Rango*

```
RuleContainer = Configuration RuleSet.

Configuration = {ConfigurationDirective}.

ConfigurationDirective =
        ('EvaluationPeriod' | 'RewardDelay' |
        'LearningRate' | 'InterferenceWeight' |
        'MinLearningRateScale' | 'FitnessWeightAppHP' |
        'FitnessWeightAppImportance' | 'FitnessWeightApp' |
        'FitnessWeightNode' | 'FitnessWeightComm' |
        'FitnessLifeSignScale' | 'FitnessCrashHealth')
         Real |
        'FitnessSicknessBoost' Real Real |
        'FitnessScope' BasicScope | 'FitnessMode' Integer |
        'SkipImpossibleActions' ('On' | 'Off') | Lag.

RuleSet = {Rule | NamedSet | NamedCondition | NamedAction}.

Rule = 'If' Condition
       'Then' Action [Lag] [Reward] [Experience] [Note].
```

A.1. Condition Production Rules

```
Condition = ConditionDefinition | ConditionName.

NamedCondition = 'DefineCondition' ConditionName ':'
                 ConditionDefintion [Note].

ConditonName = Identifier.

ConditionDefinition = SubCondition {'And' SubCondition}.

SubCondition = CardinalCondition | AverageCondition |
               MinCondition | MaxCondition.
```

```
CardinalCondition = 'Cardinal' '(' Set ')'
                    DOperator (Integer|'MaxSet' [Real]).

AverageCondition = 'Average' '('
                   AttributeAndSet DOperator Real.

MinCondition = 'MinElement' '('
               AttributeAndSet DOperator Real.

MaxCondition = 'MaxElement' '('
               AttributeAndSet DOperator Real.

Set = AppSet | NodeSet | CommSet.

AttributeAndSet = AppAttributeAndSet |
                  NodeAttributeAndSet |
                  CommAttributeAndSet.

AppAttributeAndSet = AppSet ')' AppAttribute.

NodeAttributeAndSet = NodeSet ')' NodeAttribute.

CommAttributeAndSet = CommSet ')' CommAttribute.
```

A.2. Action Production Rules:

```
Action = (ActionDefinition | ActionName).

NamedAction = 'DefineAction' ActionName ':'
              ActionDefinition [Note].

ActionName = Identifier.

ActionDefinition = ActionType ['RewardDelay' Real].

ActionType = StartApp | StopApp | RemoveApp | RelocateApp |
             RestartApp | TuneApp | CompressApp |
             SmoothPeriodExtensionApp | SchedulingMode |
             PreemptionMode | SetPriority | MonitorPeriod |
             DoNothing.

StartApp = 'Start' [SetOption] AppSet.
```

```
StopApp = 'Stop' [SetOption] AppSet.

RemoveApp = 'Remove' [SetOption] AppSet.

RelocateApp = 'Relocate' [SetOption] AppSet
              'To' [SetOption] NodeSet ['ResetTuning'].

RestartApp = 'Restart' [SetOption] AppSet
             'On' [SetOption] NodeSet ['ResetTuning'].

TuneApp = ('TunePeriod' | 'TunePriority' | 'TuneDeadline' |
           'TuneReactionTime' | 'TunePeriodAndPriority' |
           'TunePeriodAndDeadline' |
           'TunePeriodAndReactionTime') [SetOption]
           AppSet 'To' Factor.

CompressApp = 'Compress' [SetOption]
              AppSet 'To' Factor.

SmoothPeriodExtensionApp = 'Extend' [SetOption]
                           AppSet 'To' ('On' | 'Off').

SchedulingMode = 'SetScheduling' [SetOption]
                 (NodeSet 'To' Integer ['Adjust'] |
                 CommSet 'To' Integer).

PreemptionMode = 'SetPreemption' [SetOption]
                 (NodeSet | CommSet) 'To' Integer.

SetPriority = 'SetPriority' [SetOption]
              (AppSet) 'To' Factor.

MonitorPeriod = 'SetMonitoringPeriod' [SetOption]
                (AppSet | NodeSet | CommSet) 'To' Factor.

DoNothing = 'DoNothing'

Reward = 'Reward' Real.

Experience = 'Experience' Integer.

Lag = 'Lag' Integer.

Note = 'Note' String.
```

A.3. Set Production Rules

```
AppSet = (AppSetDefinition | AppSetName).

NodeSet = (NodeSetDefinition | NodeSetName).

CommSet = (CommSetDefinition | CommSetName).

NamedSet = 'DefineSet'
           (AppSetName ':' AppSetDefinition) |
           (NodeSetName ':' NodeSetDefinition) |
           (CommSetName ':' CommSetDefinition) [Note].

AppSetName = Identifier.

NodeSetName = Identifier.

CommSetName = Identifier.

AppSetDefinition = 'App' (DirectAppId |
                   Scope [AppQueryClausesList]).

NodeSetDefinition = 'Node' (DirectId |
                    Scope [NodeQueryClausesList]).

CommSetDefinition = 'Comm' (DirectId |
                    Scope [CommQueryClausesList]).
```

A.4. Support Production Rules

```
SetOption = 'All'.

Scope = BasicScope | CompositeScope.

BasicScope = 'Global' | 'System' | 'Local'.

CompositeScope = 'NonLocal' | 'NonSystem' |
                 'LocalSystem' | 'LocalNonSystem' |
                 'NonLocalSystem' | 'NonLocalNonSystem'.

DirectAppId = DirectId ['Started' | 'NonStarted']
```

```
DirectId = String ['Local'].

Unsigned = Digit {Digit}.

Integer = ['-'] Unsigned.

Real = Integer [',' Unsigned].

Identifier = Letter {Letter | Digit}.

Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
        '8' | '9'.

Letter = ('a' .. 'z') | ('A' .. 'Z') | '_' | '#'.

String = '"' {Letter | Digit | '/' | '<' | '>' |
              '!' | '?' | '=' | '(' | ')' | '*' | '+' |
              '-' | ';' | ':' | '.' | ',' | ' '} '"'.

Factor = AbsoluteValue | RelativeValueToCurrent.

AbsolutValue = Real.

RelativeValueToCurrent =
        'Current' LinearExpression [('Limit' [Real] |
        ('ExpLimitUp' | 'ExpLimitDown') Real]].

LinearExpression = ['*' Real] [('+' | '-') Real].

RelativeValueToDemand = 'Demand' LinearExpression
                        '(' [SetOption] AppSet ')'.

AppQueryClausesList = 'Where' AppClauses {',' AppClauses}.

AppClauses = AppAttribute (DOperator AppValue | MOperator).

AppAttribute = 'Importance' | 'NodeDemand' ['%']|
               'CommDemand' ['%']| 'PeriodTune' |
               'MaxPeriodTune' | 'PriorityTune' |
               'DeadlineTune' | 'ReactionTimeTune' |
               'CompressFactor' | 'MaxCompressFactor' |
               'Health' | 'NodeHealth'| 'CommHealth' |
               'HopCount' | 'AvgCommTime' |
               'LastCommTime' | 'Period' |'Priority' |
```

```
                'Deadline' | 'ReactionTime' |
                'PeriodExtension' | 'MonitoringPeriod' |
                'Id' |'LifeSignAge' |
                'NodeLifeSignAge' | 'SystemLifeSignAge'.

NodeQueryClausesList =
        'Where' NodeClauses {',' NodeClauses}.

NodeClauses = NodeAttribute
             (DOperator NodeValue | MOperator).

NodeAttribute = ('Capacity' | 'NodeCapacity' |
                 'CommCapacityBest' | 'CommCapacityWorst' |
                 'CommInCapacityBest' |
                 'CommInCapacityWorst' |
                 'CommOutCapacityBest' |
                 'CommOutCapacityWorst') ['%'] |
                 'Health' | 'CommHealth' | 'HopCount' |
                 'AvgCommTime' | 'LastCommTime' |
                 'Scheduling' | 'Preemption' |
                 'MaxScheduling' | 'MaxPreemption' |
                 'MonitoringPeriod'| 'Id' | 'LifeSignAge'|
                 'NodeLifeSignAge' | 'SystemLifeSignAge'.

CommQueryClausesList =
        'Where' CommClauses {',' CommClauses}.

CommClauses = CommAttribute
              (DOperator CommValue | MOperator).

CommAttribute = 'Capacity' ['%'] |
                'InCapacity' ['%'] | 'OutCapacity' ['%'] |
                'Health' | 'Scheduling' | 'Preemption' |
                'MaxScheduling' | 'MaxPreemption'.

DOperator = '<' | '>' | '=' | '!=' | '<=' | '>='.

MOperator = 'max' | 'min'.

AppValue = AbsoluteValue | RelativeValueToLimit.

RelativeValueToLimit = 'Limit' LinearExpression.

NodeValue = AbsoluteValue | RelativeValueToDemand |
```

```
            RelativeValueToLimit.

CommValue = AbsoluteValue | RelativeValueToDemand.
```

B. Basic Rule Set

B.1. Configuration Objectives

```
/* -- configuration ------------------------------ */
        EvaluationPeriod 0.1
        RewardDelay 2
```

B.2. Set Definitions

```
/* -- app sets ----------------------------------- */

        DefineSet LocalUnhealthyApps :
        App Local Where Health < 0, PeriodTune >= 0
        Note "Local Apps with health value
        below 0 (and not paused)"

        DefineSet LocalUnhealthyTunableApps :
        App Local Where Health < 0, PeriodTune < Limit
        Note "Local Apps with health value
        below 0 that still can be tuned"

        DefineSet GlobalUnhealthyApps :
        App Global Where Health < 0
        Note "Global Apps with health value
        below 0"

        DefineSet LeastImportantTunableApps :
        App Global Where LifeSignAge <= Limit,
        NodeLifeSignAge <= Limit,
        PeriodTune < Limit, Importance Min
        Note "The least important apps in the system which
        still can be tuned (so app and node have also
        to be alive)"

        DefineSet LeastImportantCompressableApps :
        App Global Where CompressFactor < Limit,
```

```
        Importance Min
        Note "The least important apps in the system which
        still can be compressed"

        DefineSet LeastImportantPausableApps :
        App Global Where PeriodTune >= 0,
        Importance Min
        Note "The least important apps in the system
        which still can be paused"

        DefineSet MostImportantLocalCompressedHealthyApps :
        App Local Where CompressFactor > 0, Health >= 0.6,
        Importance Max
        Note "The most important local apps that
        are compressed"

        DefineSet MostImportantLocalTunedHealthyApps :
        App Local Where PeriodTune > 1, Health >= 0.6,
        Importance Max
        Note "The most important local apps that are tuned"

        DefineSet MostImportantLocalPausedApps :
        App Local Where PeriodTune < 0,
        Importance Max
        Note "The most important local apps that
        are paused"

        DefineSet MostImportantDeadApps :
        App System Where LifeSignAge > Limit,
        PeriodTune >= 0, Importance Max
        Note "The most important system apps with expired
        life signs (an which are not paused)"

        DefineSet ForeignOrphans :
        App LocalNonSystem Where SystemLifeSignAge > Limit
        Note "Non-system apps running on our locale node
        where we have lost contact to their systems"

/* -- node sets -------------------------------- */

        DefineSet LocalUnhealthyNode :
        Node Local Where Health <= 0.1
        Note "Local node with health value of
        0.1 and below"
```

```
        DefineSet LocalHealthyNode :
        Node Local Where Health >= 0.36
        Note "Local node with health value of
        0.36 and above"

        DefineSet NonLocalSuitableNode :
        Node NonLocal Where
        Capacity % >= Demand + 0.15 (LocalUnhealthyApps),
        LifeSignAge <= Limit, Health Max
        Note "Non local alive node with enough
        capacity (leave 15% free) to take an unhealthy app"

        DefineSet
        LocalNodeWithFifoAlsoAllowsPriorityScheduling :
        Node Local Where Scheduling = 0, MaxScheduling > 0
        Note "Local node which uses FIFO scheduling,
        but also allows priority scheduling"

        DefineSet
        LocalNodeWithoutPreemptionAlsoAllowsPreemption :
        Node Local Where Preemption = 0, MaxPreemption > 0
        Note "Local node which uses no preemption, but
        also allows preemption"

        DefineSet AliveNodeWithLowestIdIsLocal :
        Node System Where LifeSignAge <= Limit,
        Id Min, HopCount = 0
        Note "Contains the alive node with the lowest
         id in the system if this is the local node"

        DefineSet SuitableNodeForMostImportantDeadApps :
        Node Global Where LifeSignAge <= Limit,
        Capacity >= Demand (MostImportantDeadApps),
        Health Max
        Note "Global alive node with enough capacity
        to take the most important dead app"

/* -- comm sets -------------------------------- */

        DefineSet LocalUnhealthyComm :
        Comm Local Where Health <= 0.15
        Note "Local comm with health value of
        0.15 and below"
```

```
        DefineSet LocalHealthyComm :
        Comm Local Where Health >= 0.5
        Note "Local comm with health value of
        0.5 and above"

        DefineSet LocalVeryHealthyComm :
        Comm Local Where Health >= 0.65
        Note "Local comm with health value of
        0.65 and above"

        DefineSet
        LocalCommWithFifoAlsoAllowsPriorityScheduling :
        Comm Local Where Scheduling = 0, MaxScheduling > 0
        Note "Local comm which uses FIFO scheduling,
        but also allows priority scheduling"

        DefineSet
        LocalCommWithoutPreemptionAlsoAllowsPreemption :
        Comm Local Where Preemption = 0, MaxPreemption > 0
        Note "Local comm which uses no preemption, but
        also allows preemption"
```

B.3. Condition Definitions

```
/* -- the conditions ---------------------------- */
        DefineCondition RelocationMightBeUseful : C
        ardinal (LocalUnhealthyApps) > 0 And
        Cardinal (LocalUnhealthyNode) > 0 And
        Cardinal (LocalUnhealthyComm) = 0 And
        Cardinal (NonLocalSuitableNode) > 0
        Note "If there are unhealthy apps on an
        unhealthy node, but comm is ok and there are
        powerful nodes available, a relocation might
         be useful"

        DefineCondition CompressionMightBeUseful :
        Cardinal (LocalUnhealthyApps) > 0 And
        Cardinal (LocalUnhealthyComm) > 0 And
        Cardinal (LeastImportantCompressableApps) > 0
        Note "If there are unhealthy apps and an unhealthy
        comm and there are still apps that can be
        compressed, compression might be useful"
```

```
DefineCondition TuningMightBeUseful :
Cardinal (LocalUnhealthyApps) > 0 And
Cardinal (LocalUnhealthyComm) > 0 And
Cardinal (LeastImportantCompressableApps) = 0 And
Cardinal (LeastImportantTunableApps) > 0
Note "If there are unhealthy apps and an unhealthy
comm and there are no more apps that can be
compressed but still apps that can be
tuned, tuning might be useful"

DefineCondition TuningUnhealthyAppsMightBeUseful :
Cardinal (LocalUnhealthyTunableApps) > 0 And
Cardinal (LocalUnhealthyComm) = 0
Note "If there are unhealthy apps that still
can be tuned while comm is healthy, tuning
might be useful (also to make room for relocation)"

DefineCondition
TuningRemainingTunableAppsMightBeUseful :
Cardinal (LocalUnhealthyApps) > 0 And
Cardinal (LocalUnhealthyComm) = 0 And
Cardinal (LocalUnhealthyTunableApps) = 0 And
Cardinal (LeastImportantTunableApps) > 0
Note "If there are unhealthy apps while comm is
healthy and there are still tunable apps left,
tuning these apps might be useful
(also to make room for relocation)"

DefineCondition PausingMightBeUseful :
Cardinal (LocalUnhealthyApps) > 0  And
Cardinal (LeastImportantCompressableApps) = 0 And
Cardinal (LeastImportantTunableApps) = 0 And
Cardinal (LeastImportantPausableApps) > 0
Note "If there are unhealthy apps and there
are no more apps that can be compressed or
tuned but still apps that can be paused,
pausing might be useful
(also to make room for relocation) "

DefineCondition
PriorityNodeSchedulingMightBeUseful :
Cardinal (GlobalUnhealthyApps) > 0 And
Cardinal
```

```
(LocalNodeWithFifoAlsoAllowsPriorityScheduling) > 0
Note "If there are unhealthy apps in the system
and our local node is on fifo scheduling but allows
also priority based scheduling, a switch to
priority based scheduling might be useful"

DefineCondition
PriorityCommSchedulingMightBeUseful :
Cardinal (GlobalUnhealthyApps) > 0 And
Cardinal
(LocalCommWithFifoAlsoAllowsPriorityScheduling) > 0
Note "If there are unhealthy apps in the system and
our local comm is on fifo scheduling but allows
also priority based scheduling, a switch to
priority based scheduling might be useful"

DefineCondition NodePreemptionMightBeUseful :
Cardinal (GlobalUnhealthyApps) > 0 And
Cardinal
(LocalNodeWithoutPreemptionAlsoAllowsPreemption)>0
Note "If there are unhealthy apps in the system
and our local node allows preemption but preemption
is not active, a switch to preemption
might be useful"

DefineCondition CommPreemptionMightBeUseful :
Cardinal (GlobalUnhealthyApps) > 0 And
Cardinal
(LocalCommWithoutPreemptionAlsoAllowsPreemption)>0
Note "If there are unhealthy apps in the
system and our local comm allows preemption but
preemption is not active, a switch to preemption
might be useful"

DefineCondition UnpausingMightBePossible :
Cardinal (MostImportantLocalPausedApps) > 0 And
Cardinal (LocalHealthyNode) > 0 And
Cardinal (LocalHealthyComm) = MaxSet
Note "If there are local paused apps and the state
of the local node and comm is quite good,
unpausing might be possible"

DefineCondition UntuningMightBePossible :
Cardinal (MostImportantLocalTunedHealthyApps) > 0
```

```
And Cardinal (MostImportantLocalPausedApps) = 0
And Cardinal (LocalHealthyNode) > 0 And
Cardinal (LocalHealthyComm) = MaxSet
Note "If there are local tuned healthy apps and
no more local paused apps and the state of the
local node and comm is quite good ,
untuning might be possible"

DefineCondition UncompressingMightBePossible :
Cardinal
(MostImportantLocalCompressedHealthyApps) > 0 And
Cardinal (MostImportantLocalTunedHealthyApps) = 0
And Cardinal (MostImportantLocalPausedApps) = 0
And Cardinal (LocalHealthyNode) > 0 And
Cardinal (LocalVeryHealthyComm) = MaxSet
Note "If there are local compressed healthy apps
and no more local tuned or paused apps and the
state of the local node and comm is good ,
uncompresssing might be possible"

DefineCondition RestartMightBeUseful :
Cardinal (AliveNodeWithLowestIdIsLocal) > 0 And
Cardinal (MostImportantDeadApps) > 0 And
Cardinal (SuitableNodeForMostImportantDeadApps) > 0
Note "If the alive node with the lowest id is the
local node and this node detects important apps
with expired lifesigns and we have node capacity,
a restart of these apps might be a good idea
(so only one node does the restart, no conflicts)"

DefineCondition
TuningToEnableRestartMightBeUseful :
Cardinal (AliveNodeWithLowestIdIsLocal) > 0 And
Cardinal (MostImportantDeadApps) > 0 And
Cardinal (SuitableNodeForMostImportantDeadApps) = 0
And Cardinal (LeastImportantTunableApps) > 0
Note "If the alive node with the lowest id is the
local node and this node detects important apps
with expired lifesigns and we have no node capacity
but still tunable apps, tuning might be
useful to create capacity"

DefineCondition
PausingToEnableRestartMightBeUseful :
```

```
        Cardinal (AliveNodeWithLowestIdIsLocal) > 0 And
        Cardinal (MostImportantDeadApps) > 0 And
        Cardinal (SuitableNodeForMostImportantDeadApps) = 0
        And Cardinal (LeastImportantTunableApps) = 0 And
        Cardinal (LeastImportantPausableApps) > 0
        Note "If the alive node with the lowest id is
        the local node and this node detects important apps
        with expired lifesigns and we have no node capacity
        and no tunable apps but still pausable apps,
        pausing might be useful to create capacity"

        DefineCondition ForeignOrphansExist :
        Cardinal (ForeignOrphans) > 0
        Note "Do foreign orphans exist once a
        foreign CPS gets lost and we still
        have remains running"
```

B.4. Action Definitions

```
/* -- the actions ------------------------------ */

        DefineAction RelocateLocalUnhealthyApp :
        Relocate LocalUnhealthyApps To NonLocalSuitableNode
        Note "Relocate one local unhealthy
        app to a suitable node"

        DefineAction CompressLeastImportantApp :
        Compress LeastImportantCompressableApps To
        Current Limit
        Note "Compress the least important
        compressabe app to its limit"

        DefineAction TuneLeastImportantApp :
        TunePeriodAndPriority LeastImportantTunableApps To
        Current Limit
        Note "Tune the least important tuneable
        app to its limit"

        DefineAction TuneLocalUnhealthyApps :
        TunePeriodAndPriority LocalUnhealthyTunableApps To
        Current Limit
        Note "Tune a local unhealthy app to its limit"
```

```
DefineAction PauseLeastImportantApp :
TunePeriod LeastImportantPausableApps To -1
Note "Pause the least important pausable app"

DefineAction PauseLeastImportantAppFast :
TunePeriod LeastImportantPausableApps To -1
RewardDelay 0.3
Note "Pause the least important pausable app
(short reward delay to quickly make room)"

DefineAction ActivatePriorityNodeScheduling :
SetScheduling
LocalNodeWithFifoAlsoAllowsPriorityScheduling To
1 Adjust
Note "Activate priority based scheduling on local
node which currently is on fifo scheduling and
assign app priorities"

DefineAction ActivatePriorityCommScheduling :
SetScheduling
LocalCommWithFifoAlsoAllowsPriorityScheduling To 1
Note "Activate priority based scheduling on local
comms which are currently on fifo scheduling"

DefineAction ActivateNodePreemption :
SetPreemption
LocalNodeWithoutPreemptionAlsoAllowsPreemption To 2
Note "Activate preemption on local node
which currently does not use preemption"

DefineAction ActivateCommPreemption :
SetPreemption
LocalCommWithoutPreemptionAlsoAllowsPreemption To 2
Note "Activate preemption on local comms
which currently do not use preemption"

DefineAction UnpauseMostImportantLocalApp :
TunePeriodAndPriority MostImportantLocalPausedApps
To Current Limit
Note "Unpause the most important local app that
is paused"

DefineAction UntuneMostImportantLocalApp :
TunePeriodAndPriority
```

```
        MostImportantLocalTunedHealthyApps To
        Current -0.5 ExpLimitDown 1
        Note "Untune the most important local app
        that is tuned down by 0.5, don't go below 1"

        DefineAction UncompressMostImportantLocalApp :
        Compress MostImportantLocalCompressedHealthyApps To
        Current -0.1 ExpLimitDown 0
        Note "Uncompress the most important local app
        that is compressed down by 0.1, don't go below 0"

        DefineAction RestartDeadApp :
        Restart MostImportantDeadApps On
        SuitableNodeForMostImportantDeadApps
        RewardDelay 0.3
        Note "Restart the most important dead app
        on a suitable alive node"

        DefineAction KillForeignOrphans :
        Remove All ForeignOrphans
        Note "Kill all foreign orphans"
```

B.5. Rule Definitions

```
/* -- the rules ---------------------------------- */

        /* -- group 1: rules for crashes -- */

        If RestartMightBeUseful Then RestartDeadApp
        Note "Rule for restarting dead apps"

        /* rules to make room for restarting crashed
        apps in overload situations */

        If TuningToEnableRestartMightBeUseful
        Then TuneLeastImportantApp
        Note "Rule to tune the least important app in case
        there is no capacity to restart a crashed app"

        If PausingToEnableRestartMightBeUseful
        Then PauseLeastImportantAppFast
        Note "Rule to pause the least important app in
        case there is no capacity to restart a crashed
```

```
app (this also includes crashed apps => the most
important apps will run)"

/* -- group 2: general rules for
overload situations -- */

If RelocationMightBeUseful
Then RelocateLocalUnhealthyApp
Note "Rule for relocation"

If CompressionMightBeUseful
Then CompressLeastImportantApp
Note "Rule for compression"

If TuningMightBeUseful Then TuneLeastImportantApp
Note "Rule for tuning"

If TuningUnhealthyAppsMightBeUseful
Then TuneLocalUnhealthyApps Lag 2
Note "Rule if we have unhealthy tunable
apps but comm is ok, lag due to race conditions"

If TuningRemainingTunableAppsMightBeUseful
Then TuneLeastImportantApp Lag 2
Note "Rule if we have unhealthy apps but comm
is ok and there are still tunable apps left,
lag due to race conditions"

If PausingMightBeUseful Then PauseLeastImportantApp
Note "Rule for pausing"

/* -- group 3: rules for scheduling issues -- */

If PriorityNodeSchedulingMightBeUseful
Then ActivatePriorityNodeScheduling
Note "Rule to fix scheduling issues on nodes"

If PriorityCommSchedulingMightBeUseful
Then ActivatePriorityCommScheduling
Note "Rule to fix scheduling issues on comms"

If NodePreemptionMightBeUseful
Then ActivateNodePreemption
Note "Rule to fix preemption issues on nodes"
```

```
If CommPreemptionMightBeUseful
Then ActivateCommPreemption
Note "Rule to fix preemption issues on comms"

/* -- group 4: rules for undoing
overload measures -- */

If UnpausingMightBePossible
Then UnpauseMostImportantLocalApp
Note "Rule for undo pausing if the
situation gets better"

If UntuningMightBePossible
Then UntuneMostImportantLocalApp
Note "Rule for undo tuning if the
situation gets better"

If UncompressingMightBePossible
Then UncompressMostImportantLocalApp
Note "Rule for undo compression if the
situation gets better"

/* -- group 5: rules to kill foreign orphans -- */

If ForeignOrphansExist Then KillForeignOrphans
Note "Rule for killing foreign orphans once a
foreign CPS gets lost and we still have
remains running"
```

C. Additional Application Specific Emergency Rules

C.1. Additional Set Definitions

```
/* -- app sets ---------------------------------- */
        DefineSet HighImportanceDeadApps :
        App System Where NodeLifeSignAge > Limit,
        Importance > 6, PeriodTune >= 0
        /* we use node life sign age for the apps since
        we strive for node failures and failures
        of long period apps might be detected lately
        by simply using life sign age */
        Note "are there dead apps of highest importance
        (steering, brake, passenger safety)"

        DefineSet LowMediumImportanceApps :
        App System Where Importance <= 6, PeriodTune >= 0
        Note "are there apps of lower importance which
        are not paused"

        DefineSet
        MostImportantLowMediumImportancePausedAppsDead :
        App System Where Importance <= 6, PeriodTune < 0,
        Importance Max, NodeLifeSignAge > Limit
        Note "get the most important of the low/medium
        importance paused apps, take those of them who
        are dead"
```

C.2. Additional Action Definitions

```
/* -- the actions ------------------------------- */
        DefineAction PauseAllLowMediumImportanceApps :
        TunePeriod All LowMediumImportanceApps To
        - 1 RewardDelay 0.3
        Note "we pause all low medium importance apps to
```

```
        make room for the high importance apps
        (we do it fast)"

        DefineAction RestartAllHighImportanceDeadApps :
        Restart All HighImportanceDeadApps On Node System
        Where LifeSignAge <= Limit, Health Max
        Note "we restart all dead high importance apps
        on the most healthy node alive"

        DefineAction RestartLowMediumImportantDeadApp :
        Restart
        MostImportantLowMediumImportancePausedAppsDead
        On Node System Where LifeSignAge <= Limit,
        Health >= 0.36
        Note "we restart a most important low/medium
        importance paused dead app on a good alive node"
```

C.3. Rule Definitions

```
/* -- the rules ---------------------------------- */
        If Cardinal (HighImportanceDeadApps) > 0
        And Cardinal (LowMediumImportanceApps) > 0
        And Cardinal (AliveNodeWithLowestIdIsLocal) > 0
        Then PauseAllLowMediumImportanceApps
        Reward 10000
        Note "if we have high importance dead apps
        then we make room as quickly as possible,
        this rule has high priority"

        If Cardinal (HighImportanceDeadApps) > 0
        And Cardinal (LowMediumImportanceApps) = 0
        And Cardinal (AliveNodeWithLowestIdIsLocal) > 0
        Then RestartAllHighImportanceDeadApps
        Reward 10000
        Note "if we have high importance dead apps
        and no more unpaused low medium importance
        apps then we restart all high importance apps,
        this rule has high priority"

        /* this rule replaces the general rule
        "If RestartMightBeUseful Then RestartDeadApp" */
        If Cardinal (HighImportanceDeadApps) <= 0
        And Cardinal
```

```
(MostImportantLowMediumImportancePausedAppsDead)>0
And Cardinal (AliveNodeWithLowestIdIsLocal) > 0
Then RestartLowMediumImportantDeadApp
Reward 10000
Note "if we have a most important low/medium
importance paused dead app, we restart it on
a good alive node, this rule has high priority"
```

D. XML Code of the Applications used in the Evaluations

D.1. Powertrain

```
<Instructions>
  <DoRequestSensorData Target="engineSensors"
    InstructionCount="10000"/>
  <WaitForSensorData Target="engineSensors"/>
  <DoLocalProcessing InstructionCount="60000"/>
  <WaitForLocalProcessing/>
  <WaitForMethodInvocation Timeout="-1"/>
  <DoMethodReturn InstructionCount="20000" Datasize="480"
    PreCondition="-1"/>
  <DoDataToActuator Target="engineActuators"
    InstructionCount="10000" Datasize="240"/>
</Instructions>
```

D.2. Stability and Drive Dynamics

```
<Instructions>
  <DoMethodInvocation Target="Brake" InstructionCount="40000"
    Datasize="480"/>
  <DoMethodInvocation Target="Steering" InstructionCount="40000"
    Datasize="240"/>
  <DoMethodInvocation Target="Powertrain"
    InstructionCount="40000" Datasize="480"/>
  <DoMethodInvocation Target="DataRepository"
    InstructionCount="40000" Datasize="480" SkipCycles="4"/>

  <WaitForMethodInvocation Target="CruiseControl" Timeout="-1"/>
  <WaitForMethodInvocation Target="DriverWarning" Timeout="-1"/>

  <!-- Return for Driver Warning -->
  <DoMethodReturn InstructionCount="20000" Datasize="480"
    PreCondition="-1"/>
  <!-- return for Cruise Control -->
  <DoMethodReturn InstructionCount="20000" Datasize="480"
    PreCondition="-3"/>
```

```
  <DoLocalProcessing InstructionCount="70000"/>
  <WaitForLocalProcessing/>

  <DoMessagePassing Target="PassengerSafety"
    InstructionCount="100" Datasize="30" />

  <WaitForMethodReturn Target="Brake" Timeout = "0.005"/>
  <WaitForMethodReturn Target="Steering" Timeout = "0.005"/>
  <WaitForMethodReturn Target="Powertrain" Timeout = "0.005"/>
  <WaitForMethodReturn Target="DataRepository" Timeout = "0.005"
    SkipCycles="4"/>
</Instructions>
```

D.3. Steering

```
<Instructions>
  <DoRequestSensorData Target="steeringSensors"
    InstructionCount="10000"/>
  <WaitForSensorData Target="steeringSensors"/>
  <DoLocalProcessing InstructionCount="160000"/>
  <WaitForLocalProcessing/>
  <WaitForMethodInvocation Timeout="-1"/>
  <DoMethodReturn InstructionCount="20000" Datasize="240"
    PreCondition="-1"/>
  <DoDataToActuator Target="steeringActuators"
    InstructionCount="10000" Datasize="128"/>
</Instructions>
```

D.4. Light

```
<Instructions>
  <DoRequestSensorData Target="lightSensors"
    InstructionCount="10000"/>
  <WaitForSensorData Target="lightSensors"/>
  <WaitForMessagePassed Timeout="-1"/>
  <DoLocalProcessing InstructionCount="25000"
    PreCondition="-1"/>
  <WaitForLocalProcessing PreCondition="-2"/>
  <DoLocalProcessing InstructionCount="20000"/>
  <WaitForLocalProcessing/>
  <DoDataToActuator Target="lightActuators"
    InstructionCount="10000" Datasize="320"/>
```

```
  <DoMethodInvocation Target="DataRepository"
    InstructionCount="40000" Datasize="480"
    SkipCycles="0"/>
  <WaitForMethodReturn Timeout = "0.25" SkipCycles="0"/>
</Instructions>
```

D.5. Passenger Safety

```
<Instructions>
  <DoRequestSensorData Target="safetySensors"
    InstructionCount="10000"/>
  <WaitForSensorData Target="safetySensors"/>
  <WaitForMessagePassed Target="StabilityAndDriveDynamics"
    Timeout = "-1" />
  <WaitForMessagePassed Target="DriverWarning"
    Timeout = "-1" />
  <DoLocalProcessing InstructionCount="500"/>
  <WaitForLocalProcessing/>
  <DoDataToActuator Target="safetyActuators"
    InstructionCount="100" Datasize = "26"
    SkipCycles="19"/>
  <DoMethodInvocation Target="DataRepository"
    InstructionCount="15000" Datasize="480"
      SkipCycles="9"/>
  <WaitForMethodReturn Timeout = "0.0125" SkipCycles="9"/>
</Instructions>
```

D.6. Driver Warning

```
<Instructions>
  <DoRequestSensorData Target="obstacleSensors"
    InstructionCount="10000"/>
  <WaitForSensorData Target="obstacleSensors"/>
  <DoLocalProcessing InstructionCount="160000"/>
  <WaitForLocalProcessing/>
  <DoMethodInvocation Target="StabilityAndDriveDynamics"
    Datasize="480" InstructionCount="40000"/>
  <DoMessagePassing Target="PassengerSafety"
    InstructionCount="100" Datasize="30" />
  <DoMethodInvocation Target="DataRepository"
    InstructionCount="40000" Datasize="480"
```

```
    SkipCycles="0"/>
  <WaitForMethodReturn Timeout = "0.125"/>
  <WaitForMethodReturn Timeout = "0.125" SkipCycles="0"/>
</Instructions>
```

D.7. Brake

```
<Instructions>
  <DoRequestSensorData Target="brakeSensors"
    InstructionCount="10000"/>
  <WaitForSensorData Target="brakeSensors"/>
  <DoLocalProcessing InstructionCount="120000"/>
  <WaitForLocalProcessing/>
  <WaitForMethodInvocation Timeout="-1"/>
  <DoMethodReturn InstructionCount="20000"
    Datasize="480" PreCondition="-1"/>
  <DoDataToActuator Target="brakeActuators"
    InstructionCount="10000" Datasize="128"/>
  <DoMessagePassing Target="Light"
    InstructionCount="40000" Datasize="32"/>
</Instructions>
```

D.8. Cruise Control

```
<Instructions>
  <DoRequestSensorData Target="obstacleSensors"
    InstructionCount="10000"/>
  <DoLocalProcessing InstructionCount="210000"/>
  <WaitForSensorData Target="obstacleSensors"/>
  <WaitForLocalProcessing/>
  <DoMethodInvocation Target="StabilityAndDriveDynamics"
    Datasize="480" InstructionCount="40000"/>
  <DoMethodInvocation Target="DataRepository"
    InstructionCount="40000" Datasize="480"
      SkipCycles="4"/>
  <WaitForMethodReturn Timeout = "0.02"/>
  <WaitForMethodReturn Timeout = "0.02" SkipCycles="4"/>
</Instructions>
```

D.9. Data Repository

```
<Instructions>
  <WaitForMethodInvocation
    Target="StabilityAndDriveDynamics" OrWithNext="y"/>
  <WaitForMethodInvocation Target="Light" OrWithNext="y"/>
  <WaitForMethodInvocation Target="CruiseControl"
    OrWithNext="y"/>
  <WaitForMethodInvocation Target="DriverWarning"
    OrWithNext="y"/>
  <WaitForMethodInvocation Target="Navigation"
    OrWithNext="y"/>
  <WaitForMethodInvocation Target="Infotainment"
    OrWithNext="y"/>
  <WaitForMethodInvocation Target="PassengerSafety"
    OrWithNext="y"/>

  <!-- return for PassengerSafety -->
  <DoMethodReturn InstructionCount="125000" Datasize="480"
    PreCondition="7"/>
  <!-- return for Infotainment -->
  <DoMethodReturn InstructionCount="125000" Datasize="480"
    PreCondition="6"/>
  <!-- return for Navigation -->
  <DoMethodReturn InstructionCount="125000" Datasize="480"
    PreCondition="5"/>
  <!-- return for DriverWarning -->
  <DoMethodReturn InstructionCount="125000" Datasize="480"
    PreCondition="4"/>
  <!-- return for CruiseControl -->
  <DoMethodReturn InstructionCount="125000" Datasize="480"
    PreCondition="3"/>
  <!-- return for Light -->
  <DoMethodReturn InstructionCount="125000" Datasize="480"
    PreCondition="2"/>
  <!-- return for StabilityAndDriveDynamics -->
  <DoMethodReturn InstructionCount="125000" Datasize="480"
    PreCondition="1"/>
</Instructions>
```

D.10. Navigation

```
<Instructions>
  <DoRequestSensorData Target="gpsSensors"
    InstructionCount="10000"/>
  <WaitForSensorData Target="gpsSensors"/>
  <DoLocalProcessing InstructionCount="240000"/>
  <WaitForLocalProcessing/>
  <DoMethodInvocation Target="DataRepository"
```

```
    InstructionCount="40000" Datasize="480"
    SkipCycles="2"/>
  <DoDataToActuator Target="displaySpeakerActuators"
    InstructionCount="10000" Datasize="6000"/>
  <WaitForMethodReturn Timeout = "0.05" SkipCycles="2"/>
</Instructions>
```

D.11. Infotainment

```
<Instructions>
  <DoRequestSensorData Target="radioAntennaSensors"
    InstructionCount="10000"/>
  <DoLocalProcessing InstructionCount="24000"/>
  <WaitForLocalProcessing/>
  <DoMethodInvocation Target="DataRepository"
    InstructionCount="40000" Datasize="480"
    SkipCycles="4"/>
  <WaitForSensorData Target="radioAntennaSensors"/>
  <DoDataToActuator Target="displaySpeakerActuators"
    InstructionCount="10000" Datasize="6000"/>
  <WaitForMethodReturn Target="DataRepository"
    Timeout = "0.02" SkipCycles="4"/>
</Instructions>
```

E. Bibliography

[Aho+06] Alfred Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Addison Wesley, 2006. ISBN: 978-0321486813.

[al18] Y.Gheraibia et al. "An Overview of the Approaches for Automotive Safety Integrity Levels Allocation." In: *Springer* (2018). DOI: https://doi.org/10.1007/s11668-018-0466-9.

[ARS15] Frederico Alvares, Eric Rutten, and Lionel Seinturier. "Behavioural Model-based Control for Autonomic Software Components". In: *International Conference on Autonomic Computing and Communication (ICAC)*. IEEE, 2015. DOI: 10.1109/ICAC.2015.31.

[Bai+21] Y. Bai et al. "ASDYS: Dynamic Scheduling Using Active Strategies for Multifunctional Mixed-Criticality Cyber-Physical Systems". In: *IEEE Transactions on Industrial Informatics* 17.8 (2021). DOI: 10.1109/TII.2020.3027645.

[Bak+16] Kacper Bak et al. "Clafer: Unifying Class and Feature Modeling". In: *Software System Modeling* 15.3 (2016). DOI: 10.1007/s10270-014-0441-1.

[Bar04] S. Baruah. "Task partitioning upon heterogeneous multiprocessor platforms". In: *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.* 2004, pp. 536–543. DOI: 10.1109/RTTAS.2004.1317301.

[BBK19] Melanie Brinkschulte, Christian Becker, and Christian Krupitzer. "Towards a QoS-Aware Cyber Physical Networking Middleware Architecture". In: *Proceedings of the 1st International Workshop on Middleware for Lightweight, Spontaneous Environments*. MISE '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 7–12. ISBN: 9781450370349. DOI: 10.1145/3366616.3368149.

[BBR09] A. Bernauer, O. Bringmann, and W. Rosenstiel. "Generic Self-Adaptation to Reduce Design Effort for System-on-Chip". In: *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), San Francisco, USA*. 2009, pp. 126–135. DOI: 10.1109/SASO.2009.41.

[BCZ05] Daniel M. Berry, Betty H. C. Cheng, and Ji Zhang. "The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems". In: *Conference on Requirements Engineering: Foundation of Software Quality*. 2005.

[BPR08] Uwe Brinkschulte, Mathias Pacher, and Alexander v. Renteln. “An Artificial Hormone System for Self-Organizing Real-Time Task Allocation in Organic Middleware”. In: *Organic Computing*. Springer, 2008. DOI: 10.1007/978-3-540-77657-4_12.

[BPS10] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. “Fuzzy Goals for Requirements-Driven Adaptation”. In: *IEEE International Conference on Requirements Engineering*. IEEE, 2010. DOI: 10.1109/RE.2010.25.

[Bri13] Brinkschulte, U. and Müller-Schloer, C. and Pacher, M. (editors). *Proceedings of the Workshop on Embedded Self-Organizing Systems, San Jose, USA*. 2013. URL: https://www.usenix.org/conference/esos13.

[Bri17] Uwe Brinkschulte. “Prototypic Implementation and Evaluation of an Artificial DNA for Self-Descripting and Self-Building Embedded Systems”. In: *Springer EURASIP Journal on Embedded Systems* (Feb. 2017). ISSN: 1687-3963. DOI: 10.1186/s13639-016-0066-2. URL: https://jes-eurasipjournals.springeropen.com/articles/10.1186/s13639-016-0066-2.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. 3rd edition. Springer New York, NY, 2011. ISBN: 978-1-4614-0675-4.

[CG12] Shang-Wen Cheng and David Garlan. “Stitch: A Language for Architecture-Based Self-Adaptation”. In: *Journal of Systems and Software* 85.12 (2012). DOI: 10.1016/j.jss.2012.02.060.

[Cou+11] George Coulouris et al. *Distributed Systems: Concepts and Design*. 5th edition. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011.

[CSI09] CSIRO. *Centre for Complex Systems*. 2009. URL: http://www.dar.csiro.au/css/.

[DAn19] D’Angelo, M. et al. “On Learning in Collective Self-Adaptive Systems: State of Practice and a 3D Framework”. In: *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2019. DOI: 10.1109/SEAMS.2019.00012.

[EKM11] Naeem Esfahani, Ehsan Kouroshfar, and Sam Malek. “Taming Uncertainty in Self-Adaptive Software”. In: *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2011. DOI: 10.1145/2025113.2025147.

[EU09] EU. *Program Future Emerging Technolgies FET - Complex systems*. 2009. URL: http://cordis.europa.eu/ist/fet/co.htm.

[FB22] Melanie Feist and Christian Becker. “A Comprehensive Approach of a Middleware for Adaptive Mixed-Critical Cyber-Physical Networking”. In: *International Conference on Pervasive Computing and Communications*. IEEE, 2022. DOI: 10.1109/PerComWorkshops53856.2022.9767312.

[Fei+22] Melanie Feist et al. “Rango: An Intuitive Rule Language for Learning Classifier Systems in Cyber-Physical Systems”. In: *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE Computer Society, Sept. 2022, pp. 31–40. DOI: 10.1109/ACSOS55765.2022.00021.

[GB15] M. García-Valls and R. Baldoni. “Adaptive Middleware Design for CPS: Considerations on the OS, Resource Managers, and the Network Run-Time”. In: *Conference on Adaptive and Reflective Middleware*. ACM, 2015. DOI: 10.1145/2834965.2834968.

[GLV13] M. Garcia Valls, I. Rodríguez Lopez, and L. Fernández Villar. “iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems”. In: *IEEE Transactions on Industrial Informatics* 9.1 (Feb. 2013), pp. 228–236. ISSN: 1551-3203. DOI: 10.1109/TII.2012.2198662.

[GPB18] Ilias Gerostathopoulos, Christian Prehofer, and Tomas Bures. “Adapting a System with Noisy Outputs with Statistical Guarantees”. In: *International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2018. DOI: 10.1145/3194133.3194152.

[Gun+14] V. Gunes et al. “A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems”. In: *KSII Transactions on Internet and Information Systems* 8 (Dec. 2014), pp. 4242–4268. DOI: 10.3837/tiis.2014.12.001.

[Guo+15] W. Guo et al. “A PSO-Optimized Real-Time Fault-Tolerant Task Allocation Algorithm in Wireless Sensor Networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 26.12 (2015). DOI: 10.1109/TPDS.2014.2386343.

[Hag18] Hani Hagras. “Toward Human-Understandable, Explainable AI”. In: *IEEE Computer* 51.9 (2018). DOI: 10.1109/MC.2018.3620965.

[HB21] E. Hutter and U. Brinkschulte. “Handling Assignment Priorities to Degrade Systems in Self-Organizing Task Distribution”. In: *International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2021. DOI: 10.1109/ISORC52013.2021.00027.

[HCZ19] B. Hu, Z. Cao, and L. Zhou. “Adaptive Real-Time Scheduling of Dynamic Multiple-Criticality Applications on Heterogeneous Distributed Computing Systems”. In: *International Conference on Automation Science and Engineering (CASE)*. IEEE, 2019. DOI: 10.1109/COASE.2019.8842895.

[Hel+04] Joseph L. Hellerstein et al. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004. ISBN: 9780471266372.

[HM08] M. Huebscher and J. McCann. “A Survey of Autonomic Computing—Degrees, Models, and Applications”. In: *ACM Computing Surveys* 40.3 (2008). DOI: 10.1145/1380584.1380585.

[Hol+00] John H. Holland et al. “What Is a Learning Classifier System?” In: *Learning Classifier Systems*. Ed. by Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 3–32. ISBN: 978-3-540-45027-6.

[HPH20] Michael Heider, David Pätzel, and Jörg Hähner. “Towards a Pittsburgh-Style LCS for Learning Manufacturing Machinery Parametrizations”. In: *Genetic and Evolutionary Computation Conference*. ACM, 2020. DOI: 10.1145/3377929.3389963.

[Hut22] Eric Hutter. “Prioritätsbasierte Taskverteilung in selbstorganisierenden Systemen”. doctoralthesis. Universitätsbibliothek Johann Christian Senckenberg, 2022, p. 325. DOI: 10.21248/gups.68742.

[IBM05] IBM. *An Architectural Blueprint for Autonomic Computing*. IBM, 2005. DOI: 10.1021/am900608j.

[Inf23] Gesellschaft für Informatik. “Petrinetze und verwandte Systemmodelle”. In: (2023). URL: https://www2.informatik.uni-hamburg.de/TGI/GI-Fachgruppe0.0.1/index.html.

[JBe+06] J.Becker et al. “Digital On-Demand Computing Organism for Real-Time Systems”. In: *Workshop on Parallel Systems and Algorithms (PASA), ARCS 2006*. Frankfurt, Germany, Mar. 2006.

[Jet89] G. Jetschke. *Mathematik der Selbstorganisation*. Harry Deutsch Verlag, Frankfurt, 1989. ISBN: 978-3-528-06346-7.

[JLS09] Z. Jaroucheh, X. Liu, and S. Smith. “A Perspective on Middleware-Oriented Context-Aware Pervasive Systems”. In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. IEEE, 2009, pp. 249–254. ISBN: 978-0-7695-3726-9. DOI: 10.1109/COMPSAC.2009.142.

[KC03] Jeffrey O. Kephart and David M. Chess. “The Vision of Autonomic Computing”. In: *IEEE Computer* (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

[Kin+18] Cody Kinneer et al. “Managing Uncertainty in Self-Adaptive Systems with Plan Reuse and Stochastic Search”. In: *International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2018. DOI: 10.1145/3194133.3194145.

[KKS12] W. Kang, K. Kapitanova, and S. H. Son. “RDDS: A Real-Time Data Distribution Service for Cyber-Physical Systems”. In: *IEEE Transactions on Industrial Informatics* 8.2 (May 2012), pp. 393–405. DOI: 10.1109/TII.2012.2183878.

[Klu+06] Florian Kluge et al. “CAR-SoC - Towards and Autonomic SoC Node”. In: *Second International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2006)*. L’Aquila, Italy, July 2006.

[Klu+08] Florian Kluge et al. “A Two-Layered Management Architecture for Building Adaptive Real-time Systems”. In: *6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008)*. Capri, Italy, Oct. 2008. DOI: 10.1007/978-3-540-87785-1_12.

[Kov+17] Andrii Kovalov et al. “Task-Node Mapping in an Arbitrary Computer Network Using SMT Solver”. In: *Integrated Formal Methods*. International Conference on Integrated Formal Methods. Ed. by Nadia Polikarpova and Steve Schneider. Vol. 10510. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 177–191. ISBN: 978-3-319-66845-1. DOI: 10.1007/978-3-319-66845-1_12.

[Kov+20] Andrii Kovalov et al. “Model-Based Reconfiguration Planning for a Distributed On-board Computer”. In: *Proceedings of the 12th System Analysis and Modelling Conference*. SAM ’20: 12th System Analysis and Modelling Conference. Virtual Event Canada: ACM, Oct. 19, 2020, pp. 55–62. ISBN: 978-1-4503-8140-6. DOI: 10.1145/3419804.3420266.

[Kru18] Krupitzer, C. et al. “Using Spreadsheet-defined Rules for Reasoning in Self-Adaptive Systems”. In: *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2018. DOI: 10.1109/PERCOMW.2018.8480283.

[Les+21] Veronika Lesch et al. “An Overview on Approaches for Coordination of Platoons”. In: *IEEE Transactions on Intelligent Transportation Systems* (2021). DOI: 10.1109/TITS.2021.3115908.

[Lip+05] G. Lipsa et al. “Towards a Framework and a Design Methodology for Autonomic SoC”. In: *2nd IEEE International Conference on Autonomic Computing*. Seattle, USA, 2005. DOI: 10.1109/ICAC.2005.61.

[LL73] C. L. Liu and James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In: *Journal of the ACM* 20.1 (1973), pp. 46–61. DOI: 10.1145/321738.321743.

[Mar21] Peter Marwedel. *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Embedded Systems. Cham: Springer International Publishing, 2021. ISBN: 978-3-030-60910-8. DOI: https://doi.org/10.1007/978-3-030-60910-8.

[MDC17] Andrei Marinescu, Ivana Dusparic, and Siobhán Clarke. “Prediction-Based Multi-Agent Reinforcement Learning in Inherently Non-Stationary Environments”. In: *ACM Transactions on Autonomous and Adaptive Systems* 12.2 (2017). DOI: 10.1145/3070861.

[Mor+16] G. A. Moreno et al. “Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation”. In: *International Conference on Autonomic Computing (ICAC)*. IEEE, 2016. DOI: 10.1109/ICAC.2016.59.

[Mor+18] G. A. Moreno et al. “Uncertainty Reduction in Self-Adaptive Systems”. In: *International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2018. DOI: 10.1145/3194133.3194144.

[MT17] Christian Müller-Schloer and Sven Tomforde. *Organic Computing – Technical Systems for Survival in the Real World*. Autonomic Systems. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-68476-5. DOI: 10.1007/978-3-319-68477-2.

[Mue+07] Gero Muehl et al. “On the Definitions of Self-Managing and Self-Organizing Systems”. In: *Communication in Distributed Systems - 15. ITG/GI Symposium*. 2007, pp. 1–11. ISBN: 978-3-8007-2980-7.

[MZR21] Sina Mohseni, Niloofar Zarei, and Eric D. Ragan. “A Multidisciplinary Survey and Framework for Design and Evaluation of Explainable AI Systems”. In: *ACM Transactions on Interactive Intelligent Systems* 11.3-4 (2021). DOI: 10.1145/3387166.

[NB08] M. Nickschas and U. Brinkschulte. “Guiding Organic Management in a Service-Oriented Real-Time Middleware Architecture”. In: *6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008)*. Capri, Italy, Oct. 2008. DOI: 10.1007/978-3-540-87785-1_9.

[NCA13] A. Noguero, I. Calvo, and L. Almeida. “A Time-Triggered Middleware Architecture for Ubiquitous Cyber Physical System Applications”. In: *International Conference on Ubiquitous Computing and Ambient Intelligence*. Vol. 7656. Dec. 2013, pp. 73–80. DOI: 10.1007/978-3-642-35377-2_10.

[Noe12] Tammy Noergaard. *Embedded Systems Architecture - A Comprehensive Guide for Engineers and Programmers*. Newnes, 2012. ISBN: 9780123821973.

[OMN23] OMNet++. “Discrete Event Simulator”. In: (2023). URL: https://omnetpp.org/.

[Onl] Oracle [Online]. *Java Remote Method Invocation - Distributed Computing for Java*. URL: https://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html (visited on 08/15/2019).

[Orl+19] S. Orlov et al. “Automatically Reconfigurable Actuator Control for Reliable Autonomous Driving Functions (AutoKonf)”. In: *10th International Munich Chassis Symposium 2019: Chassis.Tech Plus*. 2019. ISBN: 978-3-658-26435-2. DOI: 10.1007/978-3-658-26435-2_26.

[Osz+18] F. Oszwald et al. “Dynamic Reconfiguration for Real-Time Automotive Embedded Systems in Fail-Operational Context”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). May 2018, pp. 206–209. DOI: 10.1109/IPDPSW.2018.00039.

[Pfa20a] Pfannemüller, M. et al. “REACT-ION: A Model-based Runtime Environment for Situation-aware Adaptations”. In: *ACM Transactions on Autonomous and Adaptive Systems* 14.4 (2020). DOI: 10.1145/3487919.

[Pfa20b] Pfannemüller, M. et al. “REACT: A Model-Based Runtime Environment for Adapting Communication Systems”. In: *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020. DOI: 10.1109/ACSOS49614.2020.00027.

[Pro02] J. Prosise. *Programming Microsoft .NET*. Redmond, WA, USA: Microsoft Press, 2002. ISBN: 0735613761.

[PS98] F. Plášil and M. Stal. “An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM”. In: *Software - Concepts & Tools* 19.1 (Mar. 1998), pp. 14–28. DOI: 10.1007/s003780050004.

[QJP12] Nauman A. Qureshi, Ivan J. Jureta, and Anna Perini. “Towards a Requirements Modeling Language for Self-Adaptive Systems”. In: *Requirements Engineering: Foundation for Software Quality*. Springer, 2012. DOI: 10.1007/978-3-642-28714-5_24.

[Ram+12] Andres J. Ramirez et al. “Relaxing Claims: Coping with Uncertainty While Evaluating Assumptions at Run Time”. In: *Model Driven Engineering Languages and Systems*. Springer, 2012. DOI: 10.1007/978-3-642-33666-9_5.

[Rat+17] D. Ratasich et al. “A Self-Healing Framework for Building Resilient Cyber-Physical Systems”. In: *International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2017. DOI: 10.1109/ISORC.2017.7.

[RBP11] Alexander von Renteln, Uwe Brinkschulte, and Mathias Pacher. “The Artificial Hormone System - An Organic Middleware for Self-organising Real-Time Task Allocation”. In: *Organic Computing - A Paradigm Shift for Complex Systems, Springer Verlag, Basel* (2011). DOI: 10.1007/978-3-0348-0130-0_24.

[Ros+21] Lukas Rosenbauer et al. “Transfer Learning for Automated Test Case Prioritization Using XCSF”. In: *Applications of Evolutionary Computation*. Ed. by Pedro A. Castillo and Juan Luis Jiménez Laredo. Springer, 2021. ISBN: 978-3-030-72699-7.

[Rot+11] M. Roth et al. “Organic Computing Middleware for Ubiquitous Environments”. In: *Organic Computing - A Paradigm Shift for Complex Systems, Springer Verlag, Basel* (2011). DOI: 10.1007/978-3-0348-0130-0_22.

[Roy+20] D. Roy et al. “GoodSpread : Criticality-Aware Static Scheduling of CPS with Multi-QoS Resources”. In: *Real-Time Systems Symposium (RTSS)*. IEEE, 2020. DOI: 10.1109/RTSS49844.2020.00026.

[Sch+10] H. Schmeck et al. “Adaptivity and Self-Organization in Organic Computing Systems”. In: *ACM Transactions on Autonomous Adaptive Systems* 5.3 (2010). DOI: 10.1145/1837909.1837911.

[Sch05] H. Schmeck. “Organic Computing - A New Vision for Distributed Embedded Systems”. In: *8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*. Seattle, USA, May 2005, pp. 201–203. DOI: 10.1109/ISORC.2005.42.

[Smi80] Smith. “The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver”. In: *IEEE Transactions on Computers* C-29.12 (1980), pp. 1104–1113. DOI: 10.1109/TC.1980.1675516.

[ST21] Anthony Stein and Sven Tomforde. “Reflective Learning Classifier Systems for Self-Adaptive and Self-Organising Agents”. In: *International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2021. DOI: 10.1109/ACSOS-C52956.2021.00043.

[Ste+17a] Anthony Stein et al. “Interpolation in the eXtended Classifier System: An Architectural Perspective”. In: *Journal of Systems Architecture* 75 (2017), pp. 79–94. ISSN: 1383-7621.

[Ste+17b] Anthony Stein et al. “Self-learning Smart Cameras - Harnessing the Generalization Capability of XCS”. In: *International Joint Conference on Computational Intelligence*. SCITEPRESS, 2017. DOI: 10.5220/0006512101290140.

[Ste+20] Anthony Stein et al. “XCS Classifier System with Experience Replay”. In: *Genetic and Evolutionary Computation Conference*. ACM, 2020. DOI: 10.1145/3377930.3390249.

[STH16] Matthias Sommer, Sven Tomforde, and Jörg Hähner. “An Organic Computing Approach to Resilient Traffic Management”. In: *Autonomic Road Transport Support Systems*. Ed. by Thomas Leo McCluskey et al. Springer, 2016. DOI: 10.1007/978-3-319-25808-9_7.

[SW09] Olivier Sigaud and Stewart Wilson. “Learning Classifier Systems: A Survey”. In: *Soft Computing* 11.11 (2009). DOI: 10.1007/s00500-007-0164-0.

[SWM19] Stepan Shevtsov, Danny Weyns, and Martina Maggio. “SimCA*: A Control-Theoretic Approach to Handle Uncertainty in Self-Adaptive Systems with Guarantees”. In: *ACM Transactions on Autonomous and Adaptive Systems* 13.4 (2019). DOI: 10.1145/3328730.

[TH11] Sven Tomforde and Jörg Hähner. “Organic Network Control – Turning Standard Protocols Into Evolving Systems”. In: *Biologically Inspired Networking and Sensing: Algorithms and Architectures*. Ed. by Pietro Lio and Dinesh Verma. IGI, 2011. DOI: 10.4018/978-1-61350-092-7.ch002.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. 1st edition. USA: Prentice Hall PTR, 2001. ISBN: 0130888931.

[TSM17] Sven Tomforde, Bernhard Sick, and Christian Müller-Schloer. *Organic Computing in the Spotlight*. Jan. 27, 2017. arXiv: 1701.08125 [cs]. URL: http://arxiv.org/abs/1701.08125 (visited on 03/09/2023).

[TTU06] W. Trumler, T. Thiemann, and T. Ungerer. “An Artificial Hormone System for Self-organization of Networked Nodes”. In: *Biologically inspired Cooperative Computing, IFIP 19th World Computer Congress 2006, August 21-24, 2006, Santiago de Chile, Chile*. 2006. DOI: 10.1007/978-0-387-34733-2_9.

[UB17] Ryan J. Urbanowicz and Will N. Browne. *Introduction to Learning Classifier Systems*. 1st. Springer Berlin, Heidelberg, 2017. ISBN: 978-3-662-55006-9.

[UM09] Ryan J. Urbanowicz and Jason H. Moore. “Learning Classifier Systems: A Complete Introduction, Review, and Roadmap”. In: *Journal of Artificial Evolution and Applications* (2009). DOI: 10.1155/2009/736398.

[VDE03] VDE/ITG (Hrsg.) “VDE/ITG/GI-Positionspapier Organic Computing: Computer und Systemarchitektur im Jahr 2010”. In: *GI, ITG, VDE* (2003).

[Vin97] S. Vinoski. “CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments”. In: *IEEE Communications Magazine* 35.2 (Feb. 1997), pp. 46–55. DOI: 10.1109/35.565655.

[Wei+09] G. Weiss et al. “Towards Self-organization in Automotive Embedded Systems”. In: *Autonomic and Trusted Computing, Brisbane, Australia*. 2009, pp. 32–46. DOI: 10.1007/978-3-642-02704-8_4.

[Whi+10] Jon Whittle et al. “RELAX: A Language to Address Uncertainty in Self-adaptive Systems Requirement”. In: *Requirements Engineering* 15.2 (2010). DOI: 10.1007/s00766-010-0101-0.

[Whi95] Randall Whitaker. *Self-Organization, Autopoiesis, and Enterprises*. 1995. URL: https://tinyurl.com/26j85ej.

[Wil95] Stewart W. Wilson. “Classifier Fitness Based on Accuracy”. In: *Evolutionary Computation* 3.2 (1995). DOI: 10.1162/evco.1995.3.2.149.

[Woo09] Michael J. Wooldridge. *An Introduction to MultiAgent Systems*. 2nd ed. Chichester, U.K: John Wiley & Sons, 2009. ISBN: 978-0-470-51946-2.

[WS21] Alexander R. M. Wagner and Anthony Stein. “On the Effects of Absumption for XCS with Continuous-Valued Inputs”. In: *Applications of Evolutionary Computation*. Ed. by Pedro A. Castillo and Juan Luis Jiménez Laredo. Springer, 2021. DOI: doi.org/10.1007/978-3-030-72699-7_44.

[Yan+14] J Yang et al. “Task Allocation for Wireless Sensor Network using Modified Binary Particle Swarm Optimization”. In: *IEEE Sensors Journal* 14.3 (2014). ISSN: 1530437X. DOI: 10.1109/JSEN.2013.2290433.

[Yin+17] X. Yin et al. “Cooperative Task Allocation in Heterogeneous Wireless Sensor Networks”. In: *International Journal of Distrributed Sensor Networks* 13.10 (2017). DOI: 10.1177/1550147717735747.

[ZGL08] Y. Zhang, C. Gill, and C. Lu. “Reconfigurable Real-Time Middleware for Distributed Cyber-Physical Systems with Aperiodic Events”. In: *2008 The 28th International Conférence on Distributed Computing Systems*. IEEE, June 2008, pp. 581–588. DOI: 10.1109/ICDCS.2008.96.