RESILIENT ARCHITECTURES THROUGH DYNAMIC MANAGEMENT OF SOFTWARE COMPONENTS

by

Anton Dimov Hristozov

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Eric Matson, Chair

Computer and Information Technology

Dr. Eric Dietz

Computer and Information Technology

Dr. Marcus Rogers

Computer and Information Technology

Approved by:

Dr. John Springer

ACKNOWLEDGMENTS

This work has been possible due to the help from my advisor and dissertation committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

ABBREVIATIONS

AADL	Architecture Analysis and Design Language
ACSL	ANSI C Specification Language
ADL	Architecture Description Language
Antlr	Another Tool for Language Recognition
ASP	Aspect Oriented Programming
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
C&C	Component and Communication
COTS	Commercial Off-the-Shelf
CPLD	Complex Programmable Logic Device
CPS	Cyber Physical Systems
CPU	Central Processing Unit
CS	Constituent System
DOD	Department of Defense
DSA	Dynamic Software Architecture
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
EKF	Extended Kalman Filter
EMF	Eclipse Modeling Framework
FAA	Federal Aviation Administration
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCS	Ground Control Station
GPL	General Purpose Language
IBD	Interface Block Diagram
IDL	Interface Definition Language
IDE	Integrated Development Environment
IoT	Internet of Things

JSON	JavaScript Object Notation
JML	Java Modeling Language
LSS	Large Scale Systems
LTL	Linear Temporal Logic
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MBSE	Model-Based Software Engineering
ML	Machine Learning
MOF	Meta Object Facility
MTD	Moving Target Defense
OMG	Object Management Group
QVT	Query/View/Transformation
ROS	Robotic Operating System
RTA	Run-Time Assurance
RTOS	Real-Time Operating System
RV	Robotic Vehicles
SA	Software Architecture
SAT	Solvers and Tools
SoS	System of Systems
SOA	Service Oriented Architecture
SysML	System Modeling Language
TEE	Trusted Execution Environment
UAV	Unmanned Aerial Vehicles
UGV	Unmanned Ground Vehicles
ULSS	Ultra Large Scale System
UML	Unified Modeling Language
VHDL	(Very High-Speed Integrated Circuit) Hardware Design Language
XMI	XML Interchange Language

XML Extensible Markup Language

ABSTRACT

The architecture of software-intensive systems is determined by their functionality and the environment they operate in. For Cyber-Physical Systems (CPS), the environment can vary in complexity and change with time. These systems operate independently, with little or no supervision, and with little maintenance. They are expected to last for long periods, that can extend to even decades. Since their operating conditions can change over time and the requirements for the systems can evolve, it is necessary to have a non-intrusive way to instantiate newer software components as the system is operating. This can extend the usability, adaptability, and longevity of such systems.

The ability to replace software components as systems are operating can also be a way to improve software diversity, safety and security. Techniques that include redundancy and diversification of the software can become a reality only if a dedicated approach for software components update is available during run-time. The practicality of this can be achieved if component interfaces, state, and dynamic management are handled appropriately. This study explores the mechanisms, and the challenges of dynamic changes through the manipulation of software components while the system is in running. The focus is on safety-critical embedded systems, which may be resource-constrained and have real-time requirements. Improving security and reliability through fault-tolerant mechanisms is the primary goal of this work.

The study of dynamic architecture is not solely restricted to individual systems. The principles that are presented are discussed in the scope of large scale systems and system of systems. Run-time assurance is also receiving attention as part of the life of dynamic systems. Techniques such modeling and validation are explored in the same context. The work presents some experimental results and discusses possible practical approaches for designing dynamic system of systems.

1. INTRODUCTION

1.1 Purpose of the study

Our dependence on the Internet of Things (IoT) and smart sensors increases with the deployment of many of these devices in all industries. These systems are generally part of the Cyber-Physical Systems (CPS) class. They have access to their environment through sensors and are connected to a network through their cyber components. CPS can be built with limited processing power and memory and typically need to be energy efficient. A subclass of these systems are the robotic vehicles (RV) which can be mainly UAV or UGV . Since they are being deployed in different environments, this determines the large surface of potential attacks against them. These characteristics determine the challenges associated with their design, operation, and vulnerabilities. The interaction between their physical and cyber parts brings new challenges and potential exploits. Therefore the successful implementation and design of such systems will continue to challenge the current state-of-the-art and state-of-the-practice techniques.

One of the salient characteristics of CPS is their complex and evolving operating environment. This leads to interaction with a multitude of sensors and exposure to possible sensor security challenges. Another characteristic of CPS is the need for autonomous operation. The autonomous work of CPS means adaptation to changing conditions. This brings the requirement of adaptability during the course of the operation of such systems. In the cases where CPS are moving, such as Unmanned Aerial Vehicles (UAV) or Unmanned Ground Vehicles (UGV), we can expect unpredictable environmental changes. As a result of the changing conditions, different sensors and control algorithms may be needed, affecting functionality. An example could be a drone flying through different terrain and relying on different localization methods, thus requiring different software components to be used. For such circumstances, we need the ability for software adaptation and dynamic reconfiguration. Dynamic behavior is also helpful in preserving safety while combating security threats.

Most modern systems are based on some architecture that utilizes independent software components with a common mechanism for exchanging messages. An example is the widespread Autosar architecture in the automotive industry [1]. This type of architecture

provides much flexibility since components communicate with other parts of the system through messages. Unfortunately, the typical systems deployed today, including Autosar and ROS, have a static structure with predefined components and messages as well as services and actions. This makes it difficult to expand, repair or fortify existing systems by bringing new components dynamically without disrupting them. This study explores the feasibility of a new class of systems using smart components and techniques to allow for the seamless integration of such components in existing systems and in newly designed ones. The future autonomous systems will be much more dynamic and adaptive, and their reliability and safety cannot be compromised.

The study analyzes the practicality and limitations of such systems and the approach to dynamic software adaptation. The research hypotheses are tested on realistic systems and assessed on the effect of dynamism on their operational characteristics. This would require modifications to an existing autopilot and other robotic software. The resilience of the augmented capabilities need to be tested through the simulation of cyber-attacks and throughout the execution of simulated missions. Objective measures of the impact of the new approach need to be compared with the base system without any changes.

1.1.1 Class of systems

The class of systems that are part of this study are UAVs and UGVs performing autonomous missions, although the techniques can apply to any CPS, even if it is not an RV. Many of these systems offer small prototypes for research purposes that are more practical and safe. In addition, the most widespread ones have simulation models that help run continuous experiments without any fear of destroying any equipment. The class of systems we focus on is real-time, resource-constrained CPS with some portable energy storage in the form of rechargeable batteries. They are characterized as nonlinear control systems because of the nature of their electrical and mechanical design. The two major sub-classes of systems used for vehicles are holonomic and nonholonomic. A holonomic system is, for example, a UAV such as a quadcopter, characterized by the ability to move independently without any restriction at any point. On the other hand, a nonholonomic system can move in specific

directions and cannot change any direction quickly in a 3D space (for example, a UGV). A nonholonomic system is a winged aircraft, and a typical holonomic example is a helicopter or a quadcopter .

One popular autopilot system is PX4 [2]. It is used mainly for UAVs but can support other vehicles. This autopilot software is pretty attractive because it has a very extensive ecosystem and works well with other tools. For example, it can connect with mission generation tools such as MAVSDK through the Mavlink protocol [3]. The simulation environment it supports includes several options among well-supported simulators such as Gazebo and Jmavsim . It coexists nicely with QGroundcontrol for managing missions. An additional benefit of PX4 and similar systems is that the code that it executes through simulation is the same code executed in an actual vehicle except for the actuator and sensor code, which is replaced by the simulator. This guarantees that any simulation environment modifications can be tested the way they are in an actual vehicle.

The main characteristic of such systems is that they are moving in some cases in 2D or 3D space and are therefore safety-critical. This comes from the fact that they can crash and be destroyed and endanger other systems and people in the process. Because of the fact that they are moving, such systems can encounter large areas where unforeseen dangers can exist, and many new cyber attacks can be launched against them. Therefore the reliability and adaptability of such systems are crucial to their survival. A common goal for civilian or military CPS that can move is that they need to complete their mission with acceptable mission success and quality. In many cases, just completing the mission is not enough, but completing it while meeting specific criteria is the desired outcome.

We can consider that every mission has some essential priorities for its completion. Some of these priorities may be conflicting, and strategies for optimizing the mission results may be needed. Commonly used priorities can be time, survival, and mission efficiency. If some missions can be successful based on the environment just by achieving the time goal, others may be in a hostile environment where survival is paramount, and everything else is secondary. Similarly, the efficiency of the mission may refer to the distance traveled, the terrain passed, and the resources used to travel, such as fuel, for example. Defining mission utility parameters can help to perform an objective assessment of the mission quality.

The assessment of how well the mission is accomplished can be achieved through mission-validation techniques.

1.1.2 Problems that need to be solved

Numerous problems need to be resolved to make the dynamic architecture management approach practical and quantifiable. The issues stem from the fact that different software components or systems differ in size, complexity, internal state, and parallelism. Therefore, the approach needs to focus on representing the most critical aspects of any software component or system, such as interfaces, states, and contracts, including non-functional properties such as timing and memory safety. The timing issue is significant for real-time CPS, which is the class of systems on which the study focuses. The same is valid with memory safety and interactions between different parts of the system.

Software components and even systems can be designed by different vendors, and teams and can use different technologies and programming languages, and undergo different types of testing, verification, and validation. There is no universal method that guarantees the applicability of a software component in existing architectures. This comes from the fact that the testing and the operational environments may differ significantly. This could hinder integration and make the process more costly. This shortcoming is a significant hurdle in the adoption and integration and the ability to perform any run-time replacement with COTS components. A general formal approach is needed to make this task possible. This study explores the possibility of creating a framework and using a Domain Specific Language (DSL) to facilitate component interfaces and properties that can help in dynamic component management . A DSL can be very valuable in defining component interfaces explicitly. Such interfaces can then be assessed and compared through programmatic techniques and tools.

One problem when a component needs to be shut down and replaced by a new one is that it can have a significant and complex state that has been accumulated and continuously evolves as the system operates. Therefore, saving the state and being able to recreate it in a different component instance is a compelling challenge, especially if we attempt to do it in a generic way. Suppose the state is defined and given as a requirement to a component

vendor. In that case, it becomes possible to think of components as equivalent if they have the same state and interfaces, even with different implementations. This is definitely one of the great challenges in this work.

The other very significant problem that needs to be tackled is the presentation of the interface and the embedded software contract of the component or system. Replacing one component with another one that may have a very different implementation can only happen if the interfaces are preserved. The first important thing is to define what goes into defining an interface. Then how the interface is defined follows. Finally, how the interface definition is used is very important. There is no universal way to describe interfaces for off-the-shelf components that developers can use in an arbitrary architecture.

1.2 Significance of the study

With the astounding achievements in hardware and software development, the possibilities of current and future systems are continuously improving. The trend is going to continue with the adoption of AI in CPS . There is a need for continuous maintenance of such complex systems, which is challenging to meet with the current approaches. At the same time, there is a push for self-adaptive systems that can react to complex and dynamic environments. Even if these systems are designed well, they can reach a limit after their deployment and will need to be updated or repaired. The current study focuses on methods for seamless component replacement, the refresh of the state, and other techniques that can provide better reliability and longevity for a system equipped with such capabilities. These new features can save money and time for the maintenance of future systems developed with such capabilities. The approach can also reduce the number of defects due to integration issues.

Dynamic control of components that are part of the software system allows for better reactions to security threats by exploiting diversity and redundancy during run-time. Such techniques make the job of any attacker much harder and minimize the window of opportunity and the predictability of replay attacks. Another outcome of diversity and redundancy is increased reliability and overall resilience . With the increase in hardware power and mem-

ory, running more diverse software will continue to become more affordable. The solution has to come from smart software architectures taking advantage of the advances in hardware and advances in dynamic software architectures.

Finally, replacing components at appropriate moments of time can help with handling the energy consumption of a CPS . This directly affects systems that are dependent on energy harvesting or battery life and is part of the overall adaptation objective. For example, in conditions where solar energy is abundant, we may need to use a different set of components instead of when the device is working with very little or no solar energy and needs to be executing its mission frugally for energy conservation. Similarly, a newly redesigned component with energy efficiency in mind may prove crucial in systems that need to be optimized for consumption after deployment.

The significance of the dynamic refresh of components or even systems is most profound in changing environmental conditions and in the presence of potential security threats. Automatic component replacement can bring new capabilities or optimize behavior for handling environmental changes, including energy consumption. In the presence of threats, techniques that periodically turn on and off components that can execute similar tasks differently can significantly increase the time for preparation of the attack and decrease or eliminate its chances of success. Variations in time and space in how components are dynamically refreshed can further strengthen the approach, making the attacker's task even harder. All this can even be done as an upgrade option after the system is deployed.

1.3 Hypotheses

The study is based on the following hypotheses:

- Dynamic component management is an approach that can benefit the operation, security, and maintenance of software-intensive systems working in a changing environment
- Resilience of software architectures can be achieved through a shift in the design methodology to run-time mechanisms in order to allow for the creation of systems with better quality, re-usability, and adaptability.

1.4 Research Questions

Research questions:

- What kind of interfaces are necessary for smart software components and systems so that they can become suitable for dynamic reconfiguration?
- How can the state of a software component be represented, saved, and restored so that it can continue operating after it has been reloaded or replaced?
- How can software components or systems be replaced with alternative instances without interrupting the flow of operations in a typical software system?

One of the major goals of this dissertation is to propose a methodology that can help improve an existing system as well as design a new system with specific qualities. By answering the following question, we make sure that the study results are applicable to a large class of systems and not only to a specific autopilot software. Since these fundamental questions apply to widespread systems such as Autosar and ROS , the results from this research can be applied to current and future versions of systems. The findings and methodologies can be applied to an even larger class of CPS because of their generic nature.

1.5 Scope

There are many classes of software-intensive systems that will come into existence in the future. The types of systems that we concentrate on are embedded systems used for CPS. These types of systems represent small and larger devices but are generally resource-constrained compared to other software systems used in IT . To further reduce the scope and make the study more concrete, we focus mainly on autonomous systems such as UAVs and UGVs , also known as robotic vehicles (RV) . The architectures of such systems use different software and hardware platforms, but they have many similarities as far as their main hardware and system software components. The primary focus of this work is the infrastructure for software components working in the application space. This is determined by the fact that the smarts of such devices are in the application, and its ability to continue

to be resilient and adaptable comes from the characteristics of the software. The scope can also include some aspects of the existing system software, including the OS, drivers, and hypervisors, when necessary.

The results are not be only applicable to CPS , although the study focuses on such systems to provide a more specific direction. Dynamic software reconfiguration is relevant to desktop and mobile applications alike and for systems with different complexity and purpose. One can even extend the scope to devices that use hardware and special firmware, such as the one used in FPGAs and CPLDs . Narrowing the scope for CPS that includes motion and focusing on software components only does running experiments and analyzing results more manageable and gives some boundaries to the study for practical purposes.

1.6 Limitations

The study focuses on a specific class of systems, but nevertheless, some limitations need to be considered.

- One limitation is the fact that real-time systems such as UGV and UAV are safety-critical, and any software glitch can turn an executing mission into a catastrophe. Therefore the research hypotheses and tests need to be verified predominantly through simulation before trying them on a real physical system. Safety precautions need to be taken, primarily if UAVs are used for experiments, given the restrictions of where they can be flown and how accessible are special UAV testing sites. There are FAA restrictions for certain classes of UAVs, and licensed pilots are required. Smaller prototypes and simulations can therefore be more practical.

- While performing any reconfiguration of software components, there is a time dependence and sensitivity of the system on the refresh of components. This dependency needs to be further assessed, and its impact on the overall system needs to be studied and simulated. The applicability of the techniques proposed in this study may not be universal and may not be achievable for very time-sensitive components or such with very complex states. We aim to find these boundaries and propose guidelines and metrics for applying the methods.

- Some very complex components can behave like black boxes. This is true for some systems that use AI solutions. In these situations, general component refresh or update may be impractical and prohibitive to the system's health while it operates. The study starts with more manageable components and attempt to study real-world cases used in some autopilot software such as PX4 and robotic software such as the Robotic Operating System (ROS) . Simpler AI components may also be considered if they allow some form of analysis. Good possible choices are the controllers used in a UAV/UGV since they are relatively specialized and contained, and their interfaces can be described well. They also do not usually have complex states, unlike some AI solutions.

1.7 Delimitations

- The study focuses on existing autopilot systems for UAVs and UGVs. The approach needs to be general to be applied to other similar systems. The goal would be to create a universal solution based on a Domain-Specific Language (DSL) or a general Architecture Description Language (ADL) to capture component interfaces and tools to enforce them. That way, we can capture attributes for software components to quantify their interfaces and behavior. Such an approach can used to design a methodology and tools to perform experiments that can prove the theory.
- The testing can happen with existing software through simulation or on actual prototypes running the software.
- The approach is quantitative so that metrics can be compared and studied for different systems and use cases.

1.8 Assumptions

The following assumptions are considered valid for this study:

- The studied systems are real-time, resource-constrained, and safety-critical.
- These systems are adaptable in some form, even in a limited fashion, with the potential to increase their adaptability

- The studied systems are made up of software components or systems and use a message-based publish-subscribe mechanism, e.g., PX4, Ardupilot or ROS
- The used UAVs or UGVs or any other Robotic Vehicles (RV) should be able to be simulated or exist in a physical prototype
- The systems need to be open-source, and their code and documentation should be publicly available
- The developed software and documents associated with this research are made available as open-source according to the requirements of the university for such publications

1.9 Summary

This work strives to provide a comprehensive study of the subject of dynamic reconfiguration of software architectures at run-time. This includes refresh, replacement, updates, and bug fixes in existing components while the system is operating. The objective of providing a seamless operation during such dynamic changes is central to the study. Other objectives, such as increased reliability, security, and adaptability are also important, especially since the focus is on safety-critical, real-time systems. The study presents a framework, methodology, and quantitative results on the feasibility and limitations of meeting these objectives. All experiments use real-world systems to prove the approach's feasibility and quantify the findings.

2. REVIEW OF THE LITERATURE

2.1 Limitation of current systems

Most systems that control robotic vehicles are based on a design-time paradigm, where the entire architecture is well-established and fixed. Such architectures are static and designed to work in a particular way without any possibility of being changed [4]. Unfortunately, the usage of such systems is in ever-evolving environmental conditions, under different forms of threats and cyber-attacks , and in the presence of component failures and degradation throughout the life of the systems due to aging and other factors, such as temperature and humidity. This type of environment and usage necessitates a different architecture that enables dynamic instantiation of components, enabling redundancy, reliability, and fault-tolerance, which leads to better resilience .

Many systems used today employ the publish-subscribe paradigm that allows for individual components to communicate asynchronously with other components in the system [5]. This allows any component to register a message topic to which one or many other components can subscribe. Such architecture can keep the components loosely coupled where only messages or services can determine their interaction. Individual components can be designed separately with different technologies and by different vendors. Publish-subscribe is a universal mechanism and allows the addition and removal of components by letting them subscribe to messages on and off. Even when publish-subscribe is used, the existing system has predetermined components, messages, services, and actions at design time.

There are several issues about how current systems are architected and built, and it all comes from their components. One issue is that they need to be built with a straightforward way to present their interfaces, behaviors, and restrictions. This makes it difficult to replace, repair or refresh components at run-time. Another issue is that the characteristics of software components need to be described in a formal way that can be used by programmatic tools for analysis or dynamic management [6]. Many of these characteristics are embedded in the code and are thus not made explicit at a system level. This restricts the building of automation and analysis tools as the information is implicitly hidden in the implementation.

A typical system needs a way to know the dependence of components on each other and what needs to happen if a component is removed and replaced by another one while the system is operating. In other words, the architecture is not explicitly defined when the system is deployed. Due to reconfiguration , timing, overall system instability, or overall survival is hard to predict or reason about without explicit knowledge about dependencies. Therefore the failure of a specific application component can bring unforeseen circumstances that can affect the whole system without a way to perform an analysis and to have a solid recovery plan [7]. Individual components only know their state and functionality and cannot be responsible for the entire state of the system. The effects of each component's or system's failure need to be analyzed and acted upon if we desire better and more reliable systems or system of systems(SoS).

Another restriction for typical maintenance and operation is that a system upgrade typically requires a reboot of the entire system. This is needed since individual components are not entirely independent, and their interfaces and effects on the rest of the system still need to be discovered. For some systems, this type of upgrade may not be an option, and a reboot of the entire system can be a long process that can take it out for some time, affecting other system interactions. This scenario may not be an option for some applications as the incessant operation may be a desirable trait. Therefore the upgrade of individual components in flight is a necessary operation that can help achieve a smooth operation and can increase the robustness of the deployed solution [8].

The changing environmental conditions of a complex system require adaptation, which can happen through loading new and appropriate components that can work in the new environment [9]. Regular systems do not have this ability, and part of it is because of the difficulty in deploying systems with components or subsystems that have well-established functionality and interfaces and a way to describe their interactions with other components that are part of the system. For self-sufficient CPS, though, it has become pretty common to expect operation without supervision for years, and the ability to perform adaptation can benefit the product's life.

2.2 Architectures of Modern Autopilot Systems

A modern autopilot system has an architecture that consists of multiple components, as shown in figure 2.1. The components shown at the diagram's top run in separate execution units and are completely decoupled. A central piece in the architecture is the micro ORB component which handles all communication through messages and events. For real drones, the system relies on a Real-Time Operating System, which provides the execution of tasks with strict timing limits. This is the system software layer on the diagram. The system software interacts with the physical world, where actuators and sensors live.

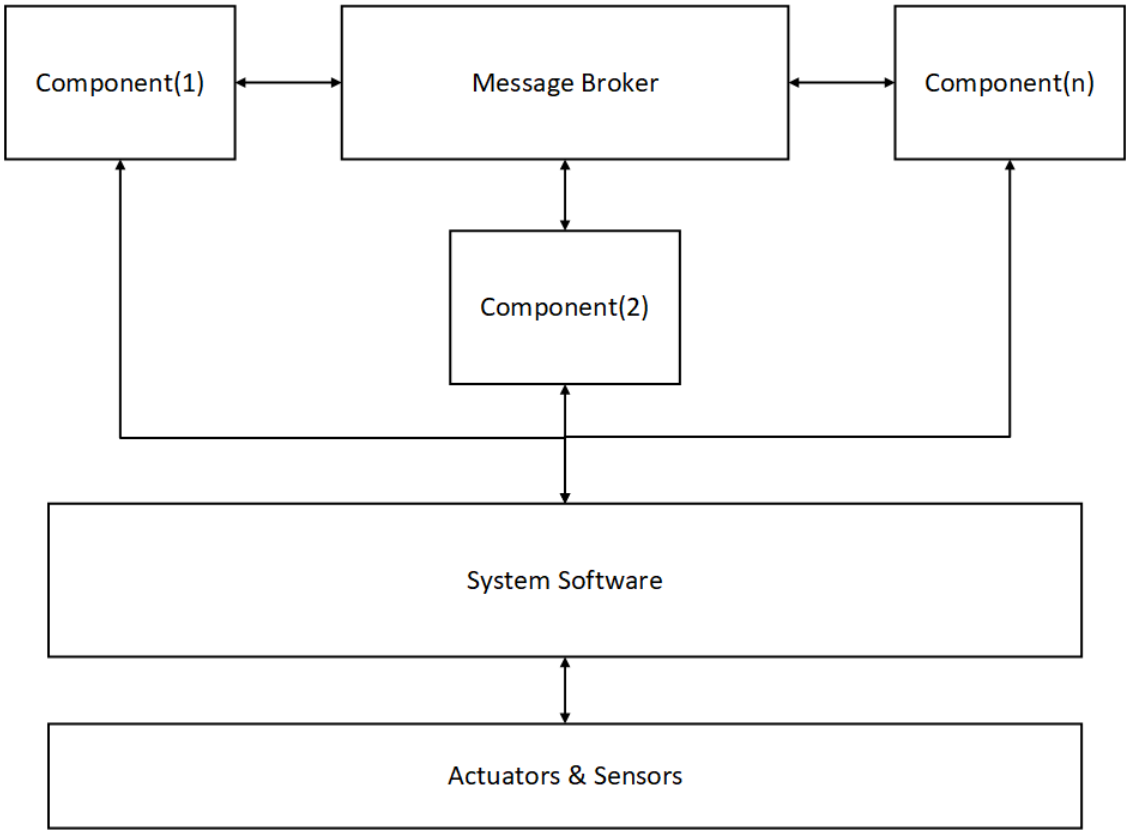

Figure 2.1. General Architecture for an Autopilot

Figure 2.2 shows the publish-subscribe communication mechanism utilized in many modern systems such as PX4, Ardupilot , and ROS. A process that wants to send messages publishes topics, and subscriber processes subscribe to one or more topics. This initialization typically happens at startup but can also happen dynamically during run-time if needed.

The publish-subscribe paradigm makes it easier to think of components as replaceable and dynamic units. As each component is free to publish as many messages, it can add new messages at run-time. Similarly, components can subscribe to new messages at run-time. For architectures that support services and actions, as is the case with ROS, the same freedom applies to services and actions.

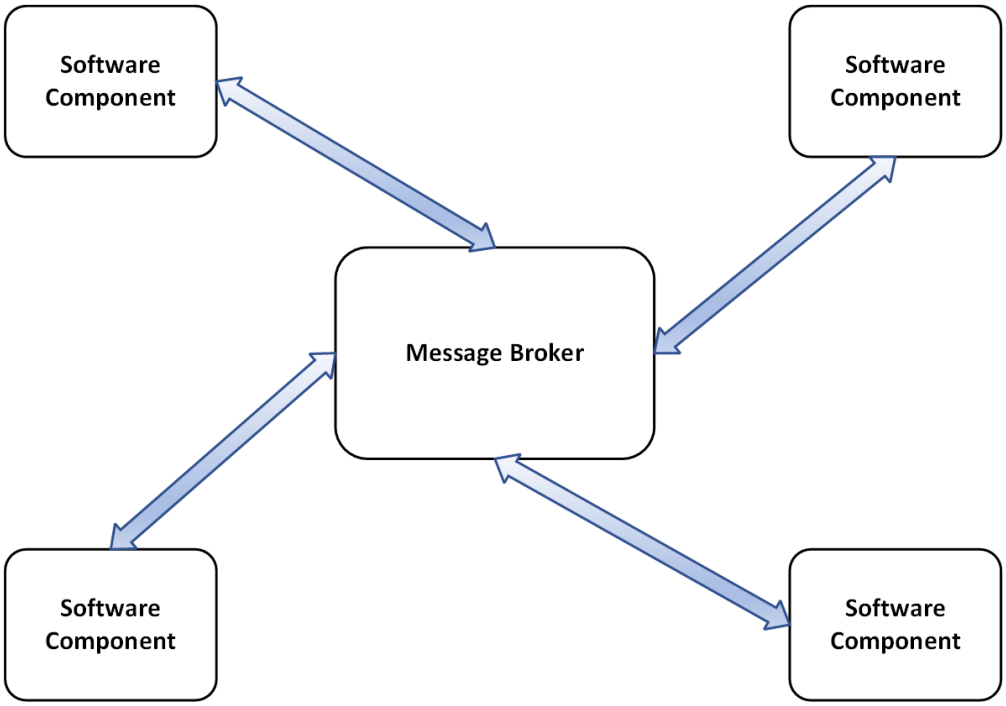

Figure 2.2. Publish-Subriber Setup

2.3 Challenges in dynamic management of software architectures

Dynamic reconfiguration at run-time is becoming a necessary feature for many different scenarios [10]. One of the main applications of dynamic reconfiguration is to adapt to changing environmental conditions. These variables can necessitate different algorithms and components with new capabilities. Such adaptation can optimize performance, energy, and overall seamless operation. Another benefit is creating more resistant architectures to cyber-attacks because of the ability to quickly change components and refresh components with compromised states. Finally, the ability to upgrade existing components by adding fixes and new features is the promise of having unattended systems that need to operate without

rebooting for years or decades. Meeting these scenarios requires overcoming the specific challenges of existing approaches.

Expressing the functional and non-functional characteristics is the first challenge that needs to be overcome to make the dynamic management of components possible [11]. The functional interfaces are better studied and easier to implement, but in real-time and safety-critical systems, the non-functional characteristics such as timing, memory safety, and self-diagnostics are more challenging to express and implement. Each component can have multiple interfaces and contracts that define its functional and non-functional behavior. The challenge needs to be addressed by defining a way to specify these component characteristics in a descriptive way that can be used programmatically and through tools. Utilizing such a domain-specific language will be one of the main focuses of this study. Describing the dynamic architecture of a system or a system of systems in an expressive and formal language can create the ability to build systems with tools and new qualities.

For the majority od systems the state is local to the individual components. The state can be modified at system initialization or at run-time [12]. Some systems can assess their state statically, although the state evolves and it is different during the course of the usage of the system. This happens because many components' internal state changes continuously [13]. Some have very many state variables, and some may have a very minimal state or be even stateless. Recovering this internal state of a component is a primary challenge because the state can have effects on the system. The difficulty in representing state universally comes from the fact that the state of a component is in flux, and the dynamic changes can be fast. In addition, components can support parallelism, making things even more complicated. This can mean that the state can be distributed throughout parallel execution contexts that are part of the component.

Software components maintain state in variables; in many cases, they can be complex objects and structures. To be able to save the state of each variable, one can use a technique called serialization [14]. The variables of an object are converted into bits- sequences and can be stored in a non-volatile memory. Deserialization is the opposite and converts the stored data from non-volatile memory to primitive types that can restore the variable of the previously serialized object. Some higher-level languages support serialization, but C++

does not support it in its standard libraries. Serialization can be achieved through specialized C++ libraries. This is important to mention since the typical autopilot software is in C or C++, and it is more difficult to use a separate language just for achieving this.

Another concern is the effects on timing when refreshing or updating components is attempted[15]. Some components can take a long long time to reach a state, where they are fully initialized, because they need to get data based on different sources. A representative component that has such characteristics is the Extended Kalman Filter (EKF) , used in many autopilots and control systems. This happens because the EKF performs sensor fusion on the data that it receives from sensors and applies algorithms that can converge after some time. Another consideration is how much time it takes for a component or a subsystem to be restarted. This can have an effect on the system's overall stability. This includes the time it takes to load and initialize a component and to update its state. This is shown in equation 2.1 where the time it takes is shown as a sum of three parts:

$$t_r = t_l + t_{\mathrm{i}} + t_s \tag{2.1}$$

where t_r is the time to refresh a component

t_l is the time to load

t_{i} is the time to initialize

t_s is the time to recover the state

Effects on the system, based on actions with individual components, are something that we need to consider. When a component is stopped and restarted, we need to consider which other components may be affected and the tolerance of absorbing this disturbance. Usually, systems are still determining what components will be affected and how because components are always running in most architectures. System-wide knowledge can be preserved by a particular subsystem that can handle these component interactions [15]. On the other hand, if components are refreshed fast enough, they may not significantly affect the system's

behavior. The implementation of approaches for such systems is going to be one of the goals of this study.

2.4 Approaches for implementing resilient systems through dynamic software component management

Architecting resilient systems needs to be a deliberate process [16]. The most important starting point is the notion of a smart component. Such components can have their diagnostics, provide interfaces and contracts that they follow, and recover and export their state. Typical software components do not have all these characteristics, affecting the overall flexibility and resilience of the architectures built with them. Performing component replacement during run-time is only possible when smart components are used.

The term component in this work has a more general meaning and can be described with the following definition: A component is a combination of software, firmware, and hardware, in some cases, allowing an independently functional unit to be part of a more extensive system. A component can have separate parallel threads of execution and an internal state and communication. It can be a single thread, a single process, or a combination of those, but the specific part is that it has a clear interface for interacting with other components as part of the system or even other systems. The emphasis on referring to a component this way is to regard its interfaces and obligations instead of the specifics of its implementation. If we extend this definition, an entire subsystem can be defined as a component.

There are some specifics when components are used in embedded systems that have real-time characteristics. The most notable ones are: a.) Components that include software can depend on hardware or be partially implemented in hardware. b.) Components need to possess strict timing characteristics. Therefore the loose coupling of components and the minimization of synchronous interfaces is crucial for this class of systems. Unique modeling languages can help with addressing the timing requirements, for example, Timber [17], which is a reactive language. Using higher-level languages has the benefit that one can model the specification closely and then proceed with implementation.

Many modeling techniques advocate a top-down approach to defining the components and interfaces. This approach is suitable when systems are first designed and may have some

significant difficulties when used with existing systems that already have millions of lines of code written. A more practical approach for existing systems is to find a way to retrofit them and enable new characteristics through modifications in their interfaces. A bottom-up approach may be necessary for these situations, starting from the key components and working on these components first and then fitting them into the existing system. This approach can preserve the properties of existing systems, add new ones, and enable dynamic changes in the life of systems retrofitted with smart components. The bottom-up approach is also very suitable for system of systems where independent systems can evolve and maintain compatibility with the rest.

2.4.1 Smart Components

Smart components include two completely separate interfaces, one for functionality and one for diagnostics and control . The diagnostics portion of the interface can be implemented independently and can live in a separate thread or a separate process so that it does not affect or be affected by the normal operation of the component. In addition, the component has a separate contract management block that handles all contract-related issues during run-time . This type of smart component allows for self-diagnostics and contract management while it works and executes its main functionality [18].

Components are sometimes classified as horizontal and vertical regarding their role in architecture. In very loosely coupled architectures, components are not in a specific hierarchy as they function according to their interfaces and can service any number of messages and services that originate from any other component. A critical characteristic of components is their suitability for reuse. The suitability for reuse can be improved significantly by keeping the interfaces and implementation of components separate. With regards to reuse, vertical components hold better promise with a higher percentage of reused functionality [19].

Special attention is applied to the way interfaces are designed. One component can have one or more interfaces, and conversely, a single interface can be used by different components. Both interface types are used in practice, significantly affecting how a component needs to be implemented. According to [19], interfaces can be stateful, meaning that they preserve

the state between invocations and can also be stateless. Stateless interfaces provide a better platform for component refresh and replacement, but stateful interfaces exist and need to be considered in this study. We will still prefer stateless interfaces as the service-oriented and message-based system interactions facilitate.

Interface adapters can augment the capabilities of an existing CPS through the enhancement of existing components. An exciting approach described in this work [20] shows how a scripting language, in this case, Lua, can be used to modify the interfaces of existing components. This approach allows for adding new functionality to existing systems by creating port monitors that can track all messages that the component receives, add functionality and make decisions about the validity of the data. Modifying existing interfaces can be used so that existing systems can be retrofitted to comply with a spec or an architecture that can allow for easier dynamic reconfiguration and can provide better reliability of the whole system.

This study will focus on the external interfaces as opposed to any internal interfaces that a software component may have [21]. An external interface can represent the component and how it fits into the existing architecture. This can be done by using a textual representation in a formal language or a simple JSON format. There will be no focus on the component's functionality as it may vary, and it is not affecting the integration of the component in the architecture. Determining if the component can fit in the architecture automatically is crucial to the success of dynamic component management.

2.4.2 Component Interactions

Since the premise of dynamic management of components is that they can be replaced or refreshed as the system is running, their effect on the entire architecture needs to be accounted for [22]. The connections between components are implicitly determined by the messages they publish and which messages they subscribe to. Similarly, components could use inputs and outputs to connect to other parts of the system. These interactions can be shown through the representation of the architecture in the form of a graph, where the vertices are the components and the edges are the connections between them [23]. This type

of representation can help find the dependencies and predict the effects of changes made to components. Such architecture is shown in figure: 2.3.

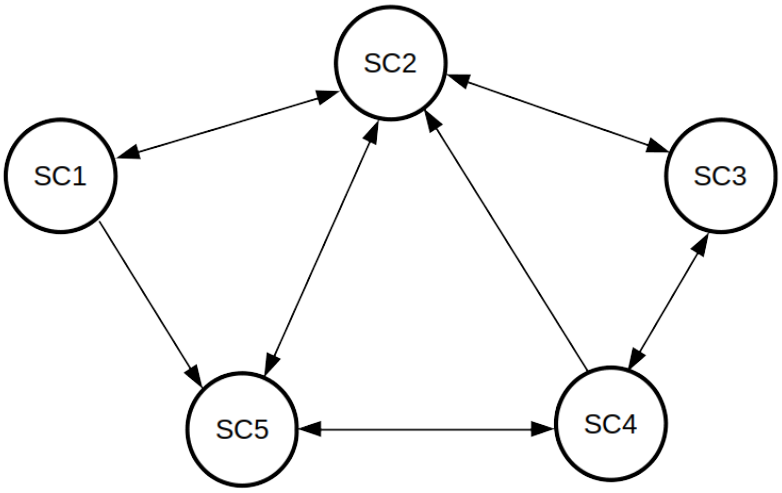

Figure 2.3. Software architecture graph showing dependencies between components

Some approaches demonstrate frameworks that can enable the replacement of components at run-time . There are a few such systems, one based on Erlang libraries and Corba [24]. The framework uses OpenRTM-aist and OPRoS, which rely on representing the state of each component and providing ports for connecting components. A special introspection interface acts as a diagnostics tool to judge if a component is in a healthy or error state and takes action. Since the approach relies heavily on Erlang libraries and tools, it is challenging to directly apply it to existing systems written predominantly in C++ and Python. Therefore a universal approach is needed that takes into consideration the architectures of existing autopilots and other robotic frameworks.

2.4.3 Component Authentication

Using software components has its risk of loading a component that is not trustworthy and can affect negatively the systems. Therefore it is necessary to have a way to ensure that components can be trusted. Component authentication and attestation need to be used before components are used. Architectures that use an attestation server and key management have been proposed [25] to handle this. A fairly popular approach exploits a shared key created during the build phase. It appends a signature to the component that can be verified only by an entity that has the same secret key. This approach is good and can be successful as long as the secret key is guarded and has sufficient length.

An HMAC approach is discussed in [26]. There a keyed hash algorithm is being used. HMAC preserves data integrity and guarantees proof of origin. As in the previous method, keeping the key secret is paramount for the approach to be secure. The overhead of checking the HMAC is a practical consideration, although it may not be an issue since it is done only once: during the initial loading of the component. Once the component is running and is in memory, then it can be trusted as we apply different techniques to guard its execution from that point on.

2.4.4 Contracts for Software Components

Contract-based software design can help with the generation of reliable components. Systems that need to provide resiliency and longevity and work in a safety-critical context need some assurances when a new component can be used [9]. This is the case because components in critical CPS can interact with physical entities and can depend on timing. In this scenario, software components' requirements must be well specified and implemented to have some guarantees during operation. Ideally, these components need to be validated and verified before use in the system.

Another motivation for contracts is the growing complexity of systems. This complexity creates difficulties in understanding the relationships and interactions between components unless these interactions are clearly defined [11]. CPS has a cyber and a physical side, which need to interact, and they may be very different. Each of these sides has different dynamics

too. Contracts can help with taming this complexity. Ultimately this can help reduce costs and schedule during the development cycle.

The notion of defining pre-conditions, invariants, and post-conditions is a simple form of defining software contracts. One can design a system with components that abide by software contracts, facilitating the system's composition from well-established building blocks. Software contracts are defined as having assumptions A and promises, also known as guarantees G [11]. The assumptions define the contract's expectations, and the guarantees are what it promises to deliver. The original work on software contracts started with the Eiffel language [27] and was adopted in some architectures over the last decades. An excellent way to approach this is to use a formal language that allows the definition of contracts during modeling.

The creation of contracts can help components to be documented better and can promote their reuse [27]. If there is a need for a component to be replaced in the future, then its contract obligations will be apparent and independent of the implementation. Therefore, the component can be easily replaced by a different one. Contracts can specify what is allowed to change and what is expected, and this limits the variability when components are implemented [28]. The specified changes in software contracts can help when components need to be upgraded or modified when some issues are discovered. This becomes a good practice for systems that need to be resilient and work without supervision for long periods.

Contracts can also help when teams from different disciplines need to interact. In many cases, engineers with various backgrounds need to implement different parts of the system [29]. This is the case when software engineers need to work with hardware engineers or control engineers. Contracts can help designs to progress concurrently and independently. The same applies to component updates and optimizations. While the contract is not violated, a component can be redesigned for better performance, for example, or better security. If the contract is preserved, the component can replace an existing one.

Contracts can be horizontal and vertical depending on their use in the system architecture. Vertical contracts define the interactions between components from different layers in the architecture, and horizontal ones help with the interaction at the same level [11]. Contracts can be used throughout the development cycle and implemented through differ-

ent design methodologies. In this respect, they can help to start from requirements up to verification and validation (V&V) .

The formal representation of contracts (C) can be described with the following pair shown in equation 2.2:

$$C = (A, G) \tag{2.2}$$

where the set of A are the assumptions and G are the promises or guarantees [11] of the contract. This definition lets us define components easier by specifying what they can guarantee and under what assumptions. Different sets of operations on A and G are also possible, and thus we can have the ability to find, for example, the union of two contracts as shown in equations 2.3, 2.4 and 2.5. This type of contract mixing allows the creation of complex contracts based on already existing ones and some logical operators.

$$C_1 = (A_1, G_1) \tag{2.3}$$

$$C_2 = (A_2, G_2) \tag{2.4}$$

$$C_1 \cup C_2 = (A_1 \cup A_2, G_1 \cup G_2) \tag{2.5}$$

Contracts can become part of the interface of components. They always have an implementation that is not specified as long as the contract requirements are met. In this respect, contracts are treated as part of the interface definition. Assumptions and guarantees can refer to functional or non-functional aspects of the design. Unlike traditional IT systems, CPS are especially dependent on non-functional requirements such as performance, timing, safety, security, and energy efficiency. Meeting some of these non-functional guarantees can be difficult and can happen with limited success.

One way to capture what components can do is to formalize their contracts. A language that contains formalism, based on Linear Temporal Logic (LTL) semantics, can capture the assumptions and guarantees with all the constraints as proposed in this work [30]. A textual or graphical representation can be used with the ability to capture system requirements for component contracts. Then the contracts can be verified through simulation. The Block

Listing 2.1. Formal Contract in LTL

```
r1 : Everytime [BRAKE_EFFORT > 0] then
[ACCELERATION < 0] within [0.2 s]
r2 : [KEY ON] implies [KEY OFF] eventually
```

Contract Language (BCL) accomplishes this through patterns that can be represented as LTL constraints. An example of a formal contract [30] can be expressed in the following example of two possible formal requirements as shown in listing 2.1. One basic non-functional assumption that needs to be guaranteed in a contract is timing. If time requirements are not met, then we cannot expect safety, security, and other requirements. This can often be achieved by using Real-Time Operating Systems (RTOS) and proper scheduling policies. Verification of timing is not easy, and it requires special tools as described here [31]. The paper's authors propose using ACSL contract language to verify what they call a timing enforcer. Timing adds a new dimension to contracts as they typically represent expected functionality from the components instead of non-functional requirements.

As discussed in [29], timing contracts are important for control systems. Traditional control theory assumes zero time for certain operations, which is not the case in real systems. That is why the authors start with the formulation of a Zero-Execution Time contract. A more realistic next step is the Bounded Execution Time contract, which guarantees that inputs are sampled, and outputs are updated before the next period of the system expires. A further step is the definition of Timing Tolerances contracts. This allows for fluctuations in certain boundaries which becomes easier to implement in software systems, especially those with no Real-Time Operating Systems (RTOS) .

Timing contracts have also been discussed in other works. The approach in [32] describes a rich component interface where the interfaces cover all functional and non-functional viewpoints. The non-functional viewpoints can be, for example, safety and security and, of course, timing. The very high-level approach for timing contracts is to specify the latency for the component and the contract associated with it. In a more detailed fashion, the timing portion of a software contract can include the period, the execution time, and jitter [33].

In order to represent timing contracts, a general approach would be to have a constraints language that can be constructed and used in the design and verification phases.

An example of such a framework is Cheddar which works well with AADL [34]. The tool allows for easy integration with other tools. It primarily focuses on verifying the temporal characteristics of real-time systems. This is achieved by expressing the system in XML format. It comes with a GUI and a simulation engine. A system can be described through processors, messages, shared resources, buffers, and tasks. Based on that, schedulability and feasibility analyses can be performed on a system even before it is implemented.

2.4.5 Software Rejuvenation

Mitigation strategies can have a limited or profound effect on the entire system. One method called software rejuvenation can be effective against attacks of different kinds that can disturb the software of autonomous vehicles. It is a form of the dynamic behavior of software architecture. The idea is that the software is restarted periodically as the system executes. This prevents the accumulation of unexpected states due to component failures or glitches or cyber attacks. The technique can be applied to systems or application software. The application in the user space is being rejuvenated, and the root of trust is in the kernel space. Other possible incarnations can be envisioned where the root of trust can be a hypervisor or a separation kernel. The most challenging issues in this method are the issues with timing and state. The time to save a checkpoint of a healthy state and recover that state needs to happen without affecting the system's operation. Once the software is recovered, then some correction of the state needs to happen so that the control software can continue smoothly to operate the vehicle.

Software rejuvenation can combat sensor, actuator, and controller attacks. The technique proposed in [35] provides for periods of shutting off communication with the outside world during software refresh and recovery of the state of the software. This reduces the probability of attacks from communication channels, for example, the GCS. Such interruption of communications provides more difficulties for the attacker too. Therefore, this method holds

promising potential for some systems, can be applied independently of the type of attacks, and does not depend on a detection mechanism.

2.4.6 Tools For Enforcement of Contracts

ACSL

Frama-C is a tool and methodology used to analyze source code written in C to provide a practical approach for verifying existing and newly developed systems. It is offered as an open-source project and has gained significant momentum in industry and research. It is based on the ACSL language, which is a formal specification language [36]. It has several plugins and is extendable. Different types of analyses are possible by using various plugins, and the addition of new ones makes the language usable for different types of applications.

The ACSL, as part of the Frama-C package, has specific primitives to express annotations in a textual format. The language supports function contracts modeled after the design by contract paradigm as introduced in Eiffel by Bertrand Meyers [29]. Function contracts provide pre-conditions, post-conditions, and loop invariants. They also define which variables can be modified by the function as part of its state and the overall program state. The loop invariant determines which properties will hold when a loop is entered and exited [37]. There are also loop assigns that show which variables are allowed to be modified by the loop. Having all these conditions specified for each function allows for rigorous analysis.

The focus of ACSL is rather specific to verifying contracts at the function level, and it is targeting the C Language. Even if it is specialized in verifying contracts for a C function, its grammar can be used and extended to create a parser that can check code used for contracts pertinent to components. This is one approach that can take an existing language and adapt it to the needs of this study. The direction can be the one taken instead of designing a domain-specific language (DSL) from scratch.

JML

Java Modeling Language (JML) is a Java-based open-source tool that focuses on an object-oriented design by contract paradigm [38]. JML, like ACSL uses special annotations

to describe the contracts. This includes the "requires" and "ensures" clauses that can model assumptions and guarantees for software contracts. There is an Antlr grammar for JML, and openjml is open source. This makes it possible to extend the JML language capabilities if needed. One valuable part of JML is that it is Java-based and can be applied to Java classes and interfaces. Using it for defining component interfaces is a possibility. The restriction that it is largely a Java-based tool may be inconvenient, but it can be overcome as the grammar can be parsed and used in various ways.

JML also supports invariants as specified by the design by contract paradigm, which could be used to ensure that a particular state is preserved. In addition, the specification supports quantifiers from Hoare's logic, such as the universal and existential quantifiers. This can help design more complex contract behavior that can express real-world scenarios. Finally, JML allows design inheritance by contract specifications to add or modify new properties without affecting original specs.

Domain Specific Languages

The proper representation of interfaces and contracts used by the components as part of a system can be achieved with a strict and more formal approach. A possible solution is to use or come up with a new Domain Specific Language (DSL) that can capture the interfaces contracts that the component uses [39]. Using a DSL can enable the use of automatic tools for integrity and run-time monitoring, component refresh and state handling. The DSL can capture the following critical elements that can help creating software architectures by using smart software components:

- Interfaces
- Contracts
- State of components

A DSL can be created by using different tools such as Antlr [40] or Xtext. Antlr can be used for creating grammars for custom languages. It can also be used to do code generation, based on the generated parser [41]. A DSL can have high-level constructs that unambiguously

define the behavior of components. Parsing the DSL constructs can produce C++ code or code for a similar language that can be used at run-time. This approach allows working with existing software components interfaces and providing necessary evolution. One popular system, such as the PX4 autopilot software as it has all the characteristics of a multi-component system with well-defined components [2]. The approach is a bottom-up approach considering analyzing an existing system and adding new properties to it. This is the opposite of the more predominant top-down paradigm through Model Driven Architecture (MDA) approaches, where the system is designed in a high-level language and is then implemented.

A simpler way to use an alternative to DSL is to use a component description language in JSON or XML format. This can be thought of as a simpler form of a full-fledged DSL, without the specifics of a formal grammar. Both approaches can represent the component's interfaces in a very precise way. This could make reuse of components and all necessary operations can be done at run-time . The interface definition of a component in association with the code that can enforce the interface can be considered an adapter attached to an existing component for the purposes described so far. Using JSON and XML is less rigorous and formal than using a DSL and, albeit easier to implement, is not going to be the preferred direction in this study.

2.4.7 Component Management and Root of Trust

Management of components to achieve dynamic refresh includes several actions, some of them discussed already. Therefore the component manager needs to be carefully designed and implemented. Its criticality is high since it determines how the whole process works. Therefore it is wise to consider the possibilities of where the component manager can reside so that it is shielded from attacks and tampering.

The main functions of the component manager subsystem are the following:

- Authenticating components and allowing only legitimate components to be used
- Handling component state
- Switching operation between components

- Loading and unloading new components
- Checking interfaces and comparing with interfaces of existing components
- Checking the health of running components through their diagnostics interface
- Maintaining dependencies graph for component interactions

The root of trust concept utilizes hardware or software implementations that are trusted and can be used to authenticate and manage other components [42]. Sometimes it is referred to as Trusted Execution Environment (TEE) . This could mean that the component manager can be placed in the kernel. This gives more protection than running in user space. It could allow for the rest of the system to run in user space. If a component in user space is compromised, then the root of the trust component manager can refresh it or replace it and maintain the system's integrity. In a more extreme scenario, the component manager can be implemented in a hypervisor and be completely protected against attacks. The approach provides an additional benefit: the component manager can be easily verified along with the hypervisor implementation because of the hypervisor's small footprint. Implementing a component manager as a root of trust component may be complex in practice and may not be feasible in this study. However, its potential benefits can be considered a method for industrial solutions.

2.5 Dynamic Run-Time Changes of System of Systems

System of systems (SoS) is an even more challenging field with respect to dynamic changes during run-time [43]. Many reasons make this challenging. SoS are formed with diverse constituent systems that have autonomy and may have been created in complete isolation. Once put together, SoS may exhibit new behavior that is unpredictable during design time [44]. What can make things worse is that with fairly different systems, it is hard to implement some common standards for development and interfaces between the different systems. Complexity is another factor that is a challenge in individual systems and an even higher challenge when they are put to work together.

The literature for SoS provides some sources that talk about dynamic reconfiguration of Large Scale Systems (LSS) or SoS, but many of the sources have rather vague approaches without some realistic examples and without precise methodologies on how to approach dynamism in general. Since the field of SoS is still an area of active research, there are many opportunities to provide practical and solid methodologies to handle dynamic reconfiguration and to analyze its effects of it. Model-Driven Development (MDD) is a promising field that also has to offer new ways of modeling the behavior of SoS instead of focusing primarily on the design of one system [45].

Dynamic changes at run-time for SoS need to support the addition or removal of a system. They also need to support the modification of a particular system. How to do this with respect to the fail-safety, security, and timing of the entire SoS is an enormous challenge. The issue with time is particularly important for real-time systems, especially hard real-time ones. Safety analysis is also very important for safety-critical SoS [46]. The same applies to security and how it can change through modifications in one system that is part of the whole assembly.

In the field of SoS complexity is an important challenge. The Model-Based Engineering approach is especially promising as a good way to tackle complexity. New modeling languages such as SysML V2 have evolved and can handle SoS [47]. Other MDE languages such as AADL also have such capabilities. Nevertheless, the issue is not trivial, and exploring how to use such tools in this domain is an interesting research direction. The challenge will be explored in later chapters of this study.

Techniques such as code generation from models can be used to speed up design and bring the ability to see the big picture through modeling and provide a bridge toward implementation. Many works focus on transforming state diagrams to source code as a proven concept. UML, SysML, AADL, and Matlab are mainly used for generating C code and, in some cases, other languages. One of the issues that need to be considered is the different levels of a semantic representation that modeling languages have compared to programming languages [48].

The majority of the efforts are focused on code generation from models. Few works look at possibilities to go in the other direction and generate models from code. Such an approach

can be useful through the evolution of systems and is a form of a bottom-up approach. Some examples include a transformation of VHDL code to SysML [49]. There are also techniques called bidirectional transformations that allow conversion from one modeling language to another, and they can be used for the transformation of production code into a model [50].

2.6 Conclusion

The process of designing fault-tolerant autonomous systems can be augmented with the help of using reusable and reliable software components or entire systems. Some of them, for example, in an autonomous vehicle, are more critical to the vehicle's performance and safety. Focusing on these components can provide better reliability and resilience to autonomous systems. Creating reliable components requires techniques such as modeling, verification , and validation . Most of the techniques and directions we discuss are used during run-time, which is more challenging than design-time techniques but has a great potential for building secure and reliable autonomous systems. The approach to express the properties of components through a DSL has excellent potential during the design and usage of complex software-intensive systems. The quality of creating systems can be improved through the help of tools that can take advantage of verification techniques and automated testing.

3. METHODOLOGY

3.1 Research Design

The research approach uses well-established software systems such as autopilot software systems for UAVs and UGVs . Some examples are well-established autopilots such as PX4 , Ardupilot , and ROS , among others. These systems are open-source, typically written in C++ or Python, and are well-known in the research community. Their popularity keeps increasing in the industry and the DOD . As a result, there are entire companies specializing in consulting and development based on these platforms because of their popularity. The limitations of such systems are well known, and there is much room for improvements that bring new research opportunities. This is particularly true for improvements concerning security and fault tolerance. Therefore they are excellent candidate platforms for this study.

The main objectives when approaching the research design are the following:

- The research should use accessible open-source software systems that are free for modification and deployment. This allows other researchers to verify the results, run the same or different experiments, and use them in their research.
- Provide an implementation of a framework that can be extended and modified by the research community.
- The research methods should be applicable in simulation and real environments.
- The results should be accessible to others through simulation environments and real physical systems such as inexpensive UAVs and UGVs .
- The approach shall be based on solid and practical engineering principles to be applied to real-life systems currently used in the industry and the DOD .

The main objective of such systems, especially the ones used by the DOD, is that they constantly strive to complete their missions successfully. Therefore it is crucial to demonstrate that the software continues to function during dynamic changes at run-time while executing a mission. This can be done by simulating realistic missions and ensuring they

are within a certain margin of error that is measured quantitatively. For this to be possible, metrics for mission quality have to be created and measured in order to be able to quantify the results of the experiments. The approach leverages techniques already used in the literature, but new ones may also be used if necessary.

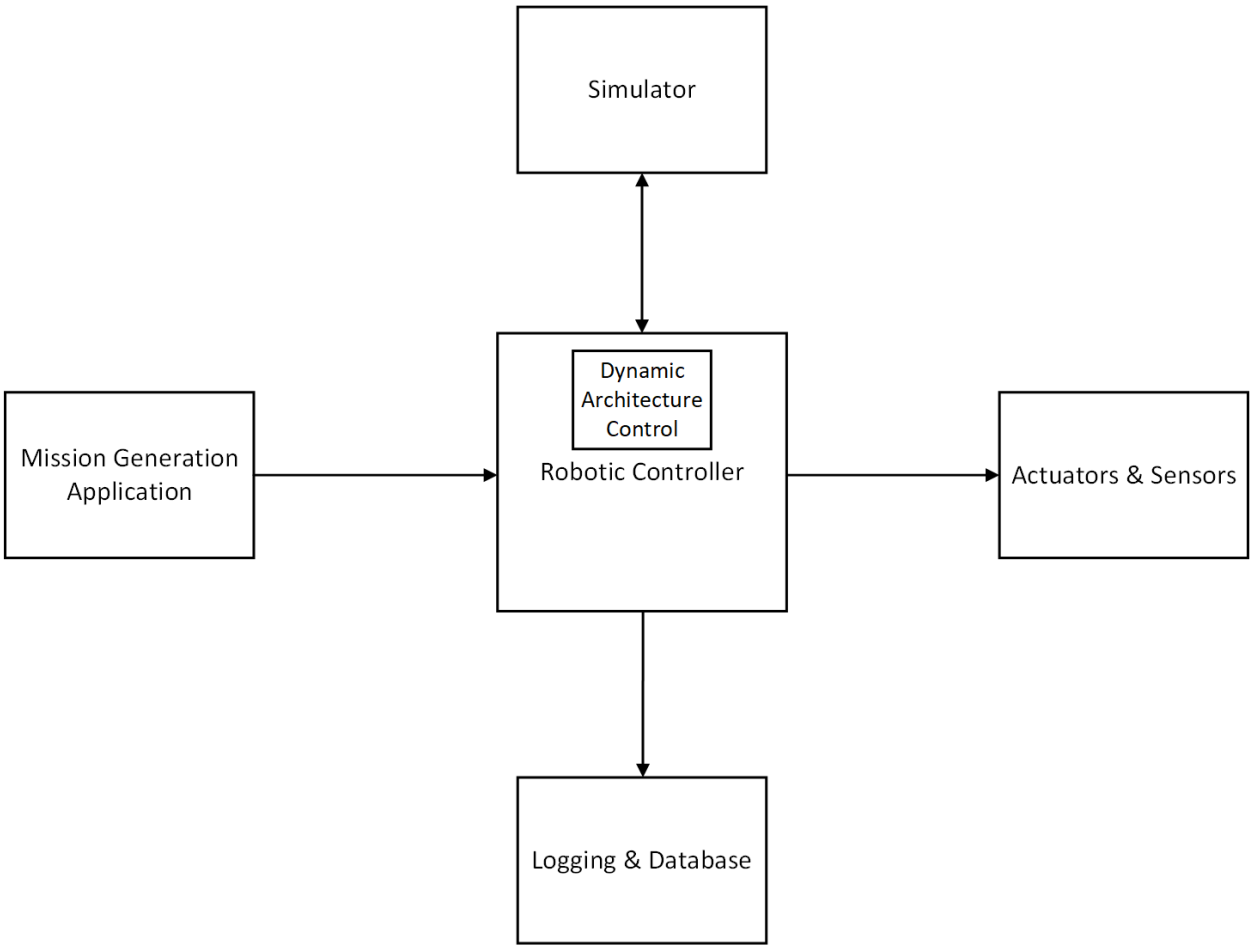

Figure 3.1. The components for setting up experiments for mission consistency. [51]

The general setup that can be used for the experiments is shown in figure 3.1. The mission controller application is used to generate predefined missions that can be used as baseline references for the experiments. The data collection and analysis subsystem is used for storing and interpreting the results. Most of the systems that are used already have extensive logging capabilities. This work's main contribution is the dynamic framework subsystem, where software components are refreshed during run-time. It provides the tools

to perform these dynamic updates during the regular operation of the system, as shown in figure 3.2.

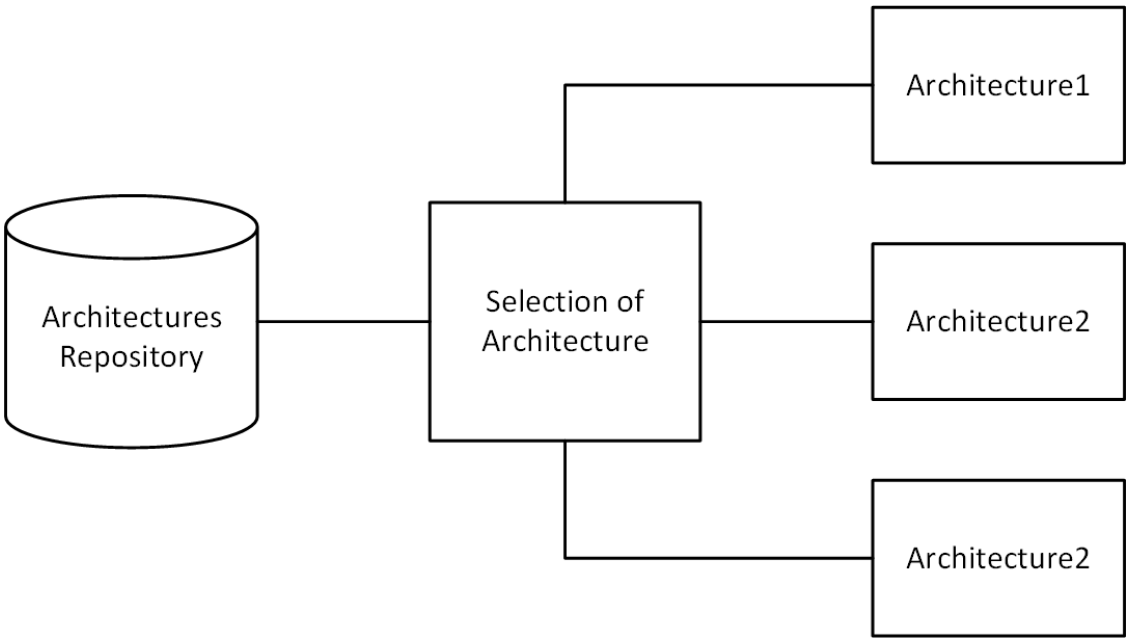

Figure 3.2. Dynamic Refresh Approach

Our primary objective is to provide a dynamic framework and methodology that is applicable to different robotic systems. Software components can be pretty different and can be implemented as software processes or threads. The focus is on the data exchanged through messages, the assumptions and guarantees for each component, and their non-functional characteristics, such as timing and memory usage. In the case of system of systems (SoS) , the interaction may happen through services as systems may be even more loosely coupled, being more diverse.

A thorough analysis of all options for experiments has discovered that from the most popular systems for controlling RVs, PX4 has the best characteristics for our study, as shown in table 3.1. PX4 has all the traits that we want in a software system. It allows for the easy addition of custom modules to extend the system. The components run in separate threads or work queues sharing a thread. It can be integrated with MAVSDK , which is a mission library. This makes the development of dedicated mission applications written in C++ or another language that can communicate with the autopilot through Mavlink

messages. The code is very generic and independent of the vehicle type. The same code that runs in simulation is the code that runs in an actual vehicle, with the exception of the actuator and sensor code, which is vehicle-specific and abstracted in the simulator. Finally, it has an excellent implementation of a publish-subscribe broker called micro ORB.

The other two popular options, Ardupilot and ROS, support some of the desired characteristics, but not all of them, just about half of them. Some compromises are required if Ardupilot or ROS are used, and therefore PX4 is the best choice for this research. It is available through git for customization and experiments like other open-source systems.

Table 3.1. Comparing Popular Architectures.

Research Architectures			
Traits / System	**PX4**	**Ardupilot**	**ROS**
Component Unit	Thread	Thread	Process/Thread
MAVSDK Support	Yes	No	No
Generic Code	Yes	No	Yes
Real Code in Sim	Yes	Yes	No
Custom Components	Yes	No	Yes
Publish/Subscibe	Yes	Yes	Yes

3.2 Procedure/methods employed to conduct the study

The study uses a quantitative approach since the data from simulations, or real missions are collected through log files that represent the mission's characteristics. The main criterion for success is the completion of the mission within certain expected limits of deterioration in the mission parameters.

The benefits of such an approach are the following:

- The information can be logged in real-time throughout the experiments and contain data regarding different variables.
- The collected information can be further analyzed through statistical and other AI techniques to extract meaning from the collected data.

- Information from multiple simulation runs can be compared and analyzed to perform analyses such as Monte Carlo for better understanding and to provide objectivity in forming conclusions.
- The information from experiments can be stored in a structured fashion to facilitate further processing and analyses.
- Regression analysis becomes possible based on similar historical data, allowing the possibility to expand the validity of experiments through modifications in the implementation of architectural changes.

The remediation techniques that are proposed in the improved architectures demonstrate the ability to erase the effects of an attack. The study can use software simulation of an attack that can use techniques to affect the system's behavior under investigation. Multiple rounds of simulation are performed in order to generalize the results and avoid the possibility of measurement errors. A Monte Carlo approach may be utilized through automation of the simulation trials to make this approach practical and easier to run. Figure 3.3 shows such a possible setup. It shows an example of a buffer overflow attack in the well-known PX4 autopilot software. The attack can be initiated from a separate software process that can send a buffer overflow message to a socket server which is added to the autopilot software. The autopilot software can then read it and unknowingly call a function that can, in turn, send a message that affects an existing software component. By compromising the component, we can replace it with a new component with a different implementation but the equivalent interface and thus return the system to a known good state.

The applicable methods to generate an attack against a software process such as an autopilot can be very different, but two of them are very popular and easier to launch. These attacks are among the ones that are also widespread not only for CPS but for other systems based on Linux OS and C/C++ implementations. The first class of attacks is the buffer overflow attack, as shown in figure 3.3, where the attacker sends more bytes than expected to a program that reads from a socket or standard input or even a file and makes it jump to an address of a function installed in the code. This malicious function can then inflict specific damage to the program it attacks. The attack uses the limitations of string

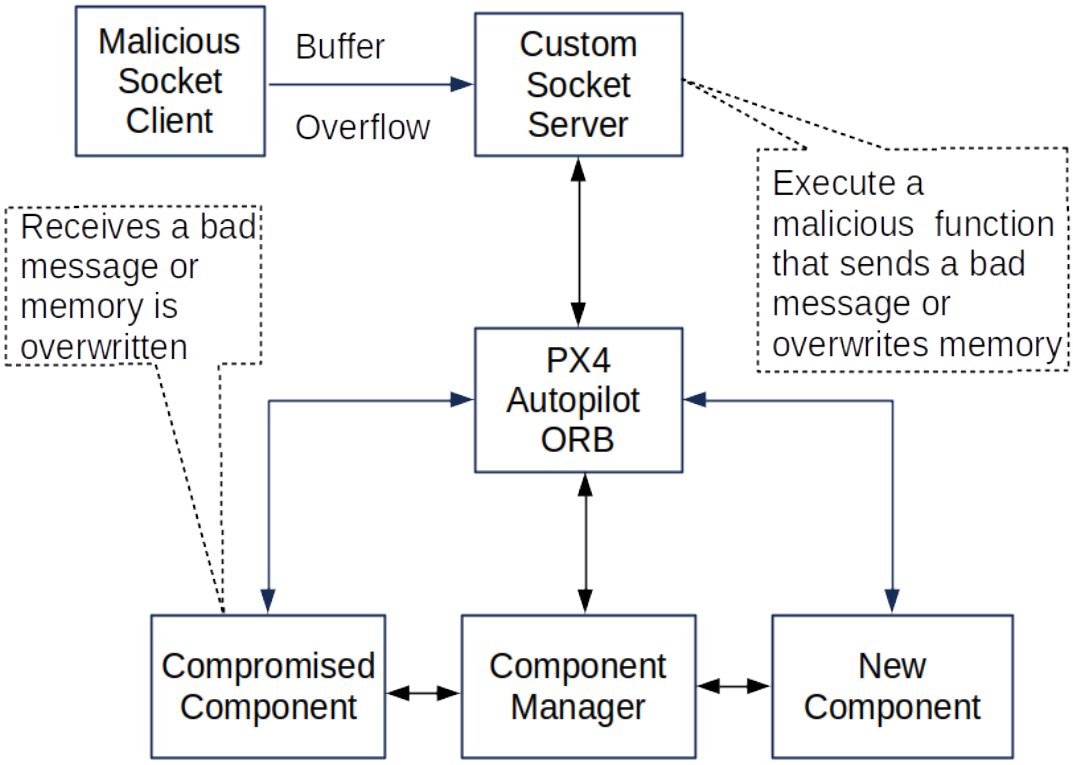

Figure 3.3. Attack Simulation [51]

representation in languages such as C and C++ to exploit the fact that the length of the string is implicit and can be used to gain an advantage. This attack setup assumes that some malicious code was loaded in advance in some fashion, possibly dynamically, and that code can be triggered later at the moment of attack.

Another scenario that can be very effective is using the /proc file system in Linux. This kind of attack does not have to install anything and can use the/proc file system's ample information. There is an entry for each running process in the /proc folder with its PID, and it can be analyzed, and modifications can be performed while the process is running. This approach makes it possible to attach each process thread and modify memory contents and file handles. The attack assumes that the attacker can find the PID of the running process and can have the correct privileges to modify the /proc entries associated with this process.

The study uses theoretical foundations for presenting contracts for software systems embedded in their interfaces. For this purpose, Hoare's logic [52] can be used. This type of logic has been widely used for several decades. The logic presents a generic approach to express relationships between inputs, outputs, and invariants. The mathematical formulation can be further implemented in a DSL to express the contracts and interfaces governing the interactions between components practically. Using such an approach can help with the validation of the implementation.

The basic formula that Hoare's logic starts with is shown in equation 3.1 where S is a program, P is an assertion true before the program is executed, and Q is an assertion that is true after the program is executed [53].

$$\{P\}S\{Q\} \tag{3.1}$$

Hoare's logic includes the following concepts that are used for verification of program correctness [52].

- Axiom of assignment

$$\{P[x := t]\}x := t\{P\} \tag{3.2}$$

- Rules of consequence

$$\frac{P \to P1, \{P1\}S\{Q1\}, Q1 \to Q}{\{P\}S\{Q\}} \tag{3.3}$$

- Rule of composition

$$\frac{\{P\}S1\{R\}, \{R\}S2\{Q\}}{\{P\}S1; S2\{Q\}} \tag{3.4}$$

- Rule of iteration

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}while\ B\ do\ S\ od\ \{P \wedge \neg B\}} \tag{3.5}$$

The above concepts are utilized in some practical implementations, such as ACSL and JML, as shown in chapter 2.

3.3 Discussion of the sample to be used in the study

The study sample uses enough experiments to get objective results representing the system's behavior in the selected testing scenarios. A typical autopilot software generates various flight parameters stored in logs. The analyses use a subset of these to prove the quality of the mission execution. The most critical data are the vehicle's actual coordinates traveled during the mission. This is usually referred to as the position during flight. In addition, parameters such as vehicle speed, acceleration, and even the derivative of the acceleration can contribute to assessing the quality of the mission execution. Another critical parameter is the time for the execution of the mission throughout different scenarios. Any degradation for various experiments are detected based on the above-mentioned basic parameters used to describe the mission execution. Some additional parameters, such as consumed energy, vibrations, and actuator variations, can also contribute to the assessment of the dynamic effects on mission quality.

3.4 Data collection procedures

The current systems' provisions used in this study can log detailed information. Some of the data are likely be in a format that is not be directly suitable for analysis. Therefore additional programmatic efforts may be needed to convert the data into a form that is suitable for further analysis. One approach is to convert the unstructured or semi-structured data into a structured form and store it in an SQL or object database. This would facilitate any further analysis and comparisons of outcomes between different experiments.

Autopilot software systems can store flight-related information in log files that can be used with visualization tools. This information can be very bulky for more extended missions, but it contains much different information for the state of the system's controllers, observers, and sensors. They have a proprietary binary format but can be analyzed with existing tools. This allows for doing before and after comparisons of data collected in logs before any changes and after specific changes are implemented in the software.

3.5 Data analysis procedures

Data analysis is performed in two main ways: offline based on the data collected and online while the experiments are performed. We need to react to specific parameters while the system is executing to make real-time decisions. Simulators and direct feedback from the performance of physical prototypes further aid the analysis. Some popular simulators that are useful in this research are JMavsim and Gazebo. The reason for selecting these two simulators is that they are integrated with PX4. Gazebo also supports ROS systems. In addition, it allows for modifications through plugins which adds flexibility for running experiments. The majority of the analyses of the logged data can be done offline. Various statistical tools and machine learning techniques can be used for these analyses. Scripting languages, such as Python and R, may be utilized because of their rich access to data processing and visualization libraries. Other tools, such as Matlab and Modelica, can also be used for data processing and visualization. In the case of analyzing PX4 logs, some web-based utilities exist, such as Flight Review. They provide numerous plots of flight data that can present valuable information for data analysis.

3.6 Conclusion

The approach in this work is based on quantitative techniques and experimental setups using well-established open-source platforms. The developed software, tools, and documents made available through source control repositories such as git. The objective is to answer the research questions. A secondary objective is to make the findings valuable and reproducible to researchers and make further investigations in this area possible. Using analyses based on machine learning and statistical techniques can help understand the data and how well the research objectives are accomplished. Using predominantly quantitative techniques facilitates this approach and make the conclusion objective and based on actual data from the experiments. Such a decision can also facilitate the comparison of the fitness of techniques and their quality in executing vehicle missions.

4. RESILIENT ARCHITECTURES THROUGH DYNAMIC RECONFIGURATION

Resilience is a non-functional requirement that, like many non-functional requirements, is hard to define and realize in practice through standard design approaches. Resilient systems are recognized as something that is very relevant today as system engineering evolves and complexity and security concerns grow. There are many different definitions of resilience in the literature. One definition of resilience is the following: "the capability of a system with specific characteristics before, during and after a disruption to absorb the disruption, recover to an acceptable level of performance, and sustain that level for an acceptable period of time " [54], p.2. This means that a system is expected to continue to execute its mission and is ultimately successful. This chapter focuses on achieving resilience through dynamic architectural changes at run-time.

The CPS work in harsh conditions and are expected to last long without much supervision. This could be years or even decades in an environment that may deviate from the expected conditions envisioned during the design of these systems. As these conditions change, the software needs to be resilient to the changes and to continue operation. This requires systems to be designed with resilience and adaptability in mind. The environment can change due to intentional security threats or gradual changes due to other factors. An utterly static architecture has limitations in how to adapt to such changes. Thus the assumption that dynamic changes are needed makes sense for both modern and future systems.

A resilient system needs to survive disruption and recover from it in a way so that it can continue operating at the expected capacity. It is also important to note that the recovery must be within certain time limits to be practical. Resilience can be achieved through different approaches. One of them is the redundancy of software entities. Many approaches exploit some sort of dynamic behavior at run-time that may have been planned during the design phase but are executed at run-time. Another one is the diversity of solutions for the same task. Diversity in time and space is a promising direction for generating resilient solutions.

Implementing resilience in a system can happen with methods that work at design-time and run-time. The design-time methods are well established and can rely on modeling techniques and formal methods. One issue with design-time techniques is that the problem space can become huge when trying to cover all possible cases. This type of design approach can quickly become intractable and very expensive. The design-time approaches can still be useful for smaller systems with smaller state spaces and minimal interactions. Unfortunately, many contemporary systems are fairly large and complex to be fully analyzable during design-time.

Methods such as fault tree analysis and end-to-end timing analysis can be very useful design-time techniques for resilient systems once the domain is well understood. Modeling languages such as AADL [55] can be used successfully to help these design efforts. Some of the modeling languages, like AADL, support the notion of modes, another example of dynamic behavior that can be conceived at design-time but used at run-time. The difficulty in getting this right is that often the environment cannot be modeled precisely, giving the designer a partial view of the system's operating conditions. In such situations, run-time methods provide more flexibility and can solve different problems with respect to resilience and flexibility.

Run-time methods allow for reactions to environmental changes, including intentional attacks on the designed system. They can also be proactive instead of reactive, making the system less prone to external disruptions. The run-time techniques can, therefore, very well complement the design-time ones. Run-time solutions are emerging as a more recent area of research, and that is why they are covered in more detail in this work. They rely on dynamic architectural solutions like the ones that will be presented in this chapter. The challenges with run-time techniques are to provide assurances for the designed systems.

4.1 Prerequisites for Dynamic Architectures through Component Updates

Components and their connections can represent software architectures. This allows for the representation of the architecture through graphs, as some researchers have suggested [56]. Different possible combinations between components can then be assembled dynami-

cally, as shown in figure 4.1. The figure demonstrates how three different architectures can be formed using a subset of the nine components shown on the right. The possible combinations can be very high, even for systems with a small number of components. A practical approach is to select the pre-validated selected subset of working scenarios that can respond to certain system properties and are sufficient to provide the desired behavior in practice. Such selected architectures can be stored in a database and can be turned on when the right conditions occur. This could happen when an internal or external event triggers mandating a different response in order to provide for different behavior. This has a direct effect on how resilient a system can be compared to a system with a static architecture. The approach allows for adding new graph configurations in case the system is enhanced or needs to be modified to work better for the existing requirements.

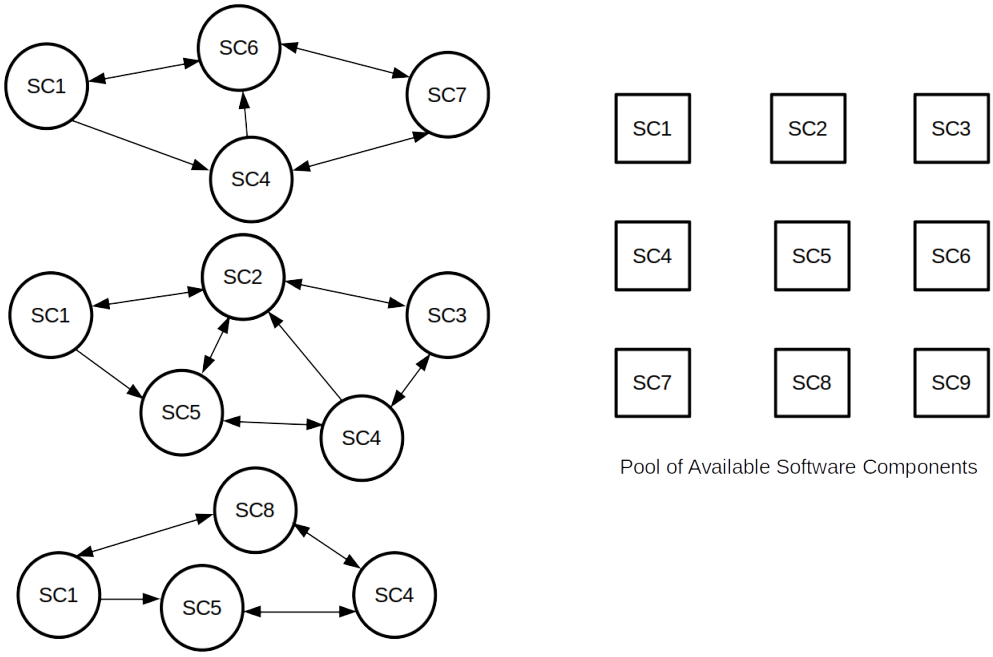

Figure 4.1. Software Architecture Configurations [57]

The prerequisites that are needed for dynamic architectures to be feasible can be presented explicitly in the following way. We can assume that systems are composed of components that have well-defined interfaces and behavior. Some architectural frameworks have attempted to come up with standards for interfaces when specifying components, for example, Autosar [1], although such standards are domain specific and are not universal. For

the purposes of realizing dynamic behavior, the interfaces have to contain certain parts that can help interchange components as the system operates. As discussed earlier, we can refer to such components as smart components. Smart components do not just have input and output messages but instead have other standard features between systems and architecture. Such an attempt to standardize the interface of a smart component is presented here. Components can have the following interface characteristics as shown in figure 4.2 [58]. Their interfaces can be represented generically through the following equation 4.1 [59]. Some of the elements listed in the proposed generic interface are influenced from well-established robotic architectures such as ROS .

$$CI = M + N + S + A + P + \Sigma + \Delta \tag{4.1}$$

, where CI represents the component's interface,

M - the set of subscribed messages

N - the set of published messages

S - the set of services

A - the set of actions

P - the set of configuration parameters

Σ - the set of state variables

Δ - the set of diagnostics services

The main contributions when defining a component interface this way are that there is a separate diagnostics and control block Δ, a state part Σ and a configuration part P. These are essential for the dynamic management of components and are typically missing in COTS components. The diagnostic block allows for a component's state to be actively controlled and monitored. The state Σ can be used for preparing a different component to start working right away when an original component is replaced. The overhead of creating such a rich interface representation allows for new opportunities that regular software components cannot provide without significant rework.

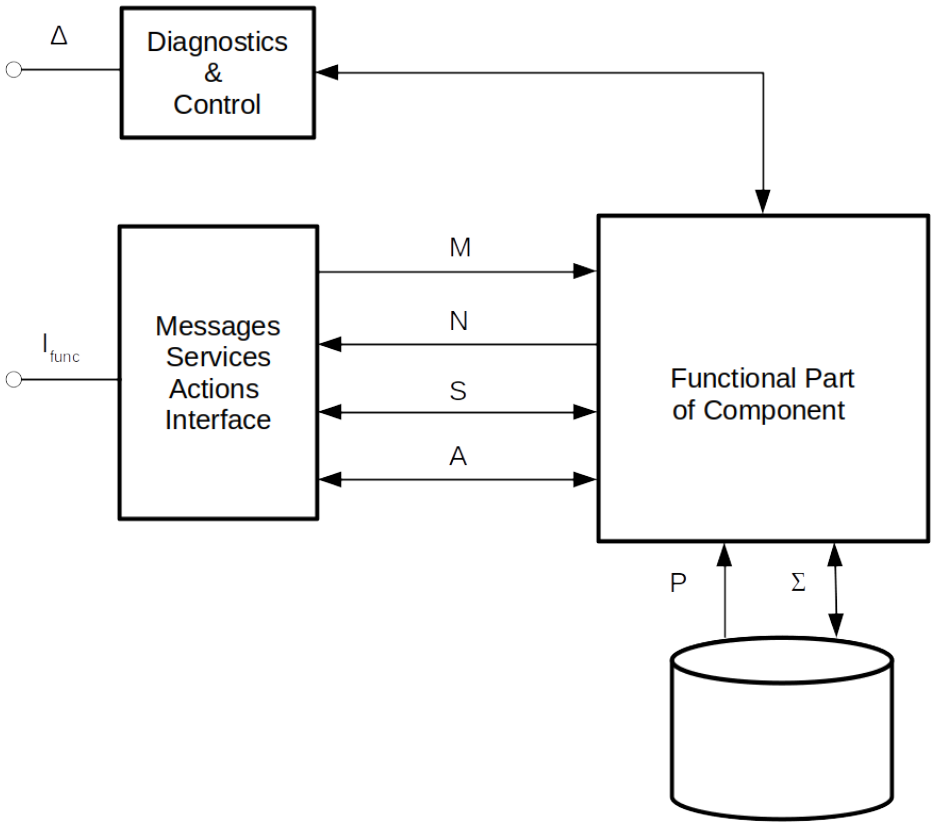

Figure 4.2. Smart component with interfaces [58]

In addition to the interface, components should not be blindly trusted as they can come from different sources. An authentication scheme of some kind should be added so that each component is checked for authenticity before it is loaded in memory and becomes part of the system. Different schemes are possible, and some of them can be fairly elaborate. A public key versus shared key mechanisms can work as the decryption's overhead will happen only once when the component is loaded. Attaching a hash or some other type of security mechanism can happen when the component is produced by a reliable source and can then be verified by the architecture at run-time.

In order for dynamic changes to happen, one needs a subsystem that manages all dynamic interactions. Such subsystem can be called component manager , and it can possess all the rules for the possible configurations that can happen when certain events occur. The different configurations can be kept in a database representing the architectures through graphs. The component manager can act when it is safe to make dynamic reconfiguration of the system. It can also qualify if components are compatible and perform authentication as they are loaded. In the case of system of systems the components will be replaced by entire systems, and there may not be a central authority to manage them. This is explored in later chapters of this work as a system of systems presents different types of challenges for dynamic management.

4.2 Management of Dynamic Architectures

Previously, it was discussed that architecture could be represented as a graph with the nodes representing the components and the edges, the actual communication channels. Given that the interface of a smart component 4.2 contains the subscribed messages M and the published message N, then each component's connections are stored in the component's interface and do not need to be explicitly associated with the connections. This is because the publish-subscribe paradigm takes care of the connections between components implicitly through its message broker . This makes dynamic rearrangements of components even easier with connections being created independently with respect to the participating components.

As described in previous work, the management of dynamic architectures can handle three general cases [57] as shown in 4.3, 4.4 and 4.5. The first case is the simplest to handle

and shows how a single component can be replaced by another component in different places of the architecture. The second case shows how one component can be replaced by more than one or when a single component can replace several components. The last case is the most general one, showing how a completely orthogonal architecture can be formed with no partial connections. Each of these cases imposes different timing and energy requirements as the system is dynamically reconfigured. The component manager subsystem needs to handle all these cases if feasible for a particular system.

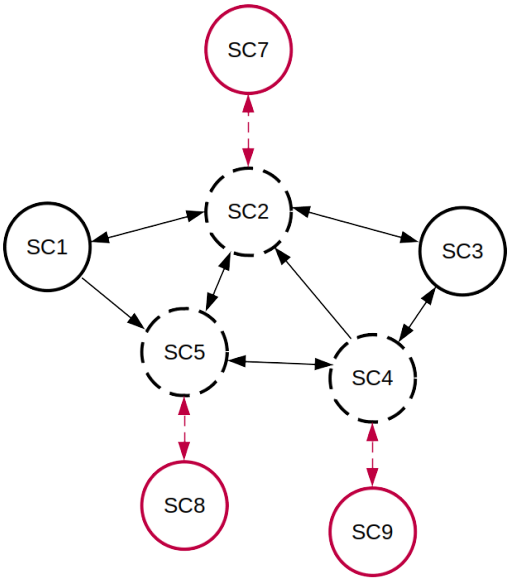

Figure 4.3. Dynamic Software Architecture - Case 1 [57]

4.3 Dynamic Architectures Incarnations

There have been some dynamic architectures helping build resilient platforms for some time. Many of them stem from the popular simplex architecture 4.6. The idea for this classic architecture is straightforward yet powerful. Every control system has a controller that, in some cases, may be fairly complex. Some of these controllers use complex math to solve differential equations in real time to generate the right control output. A problem can arise if this complex controller is wrong about the output based on a math exception or based

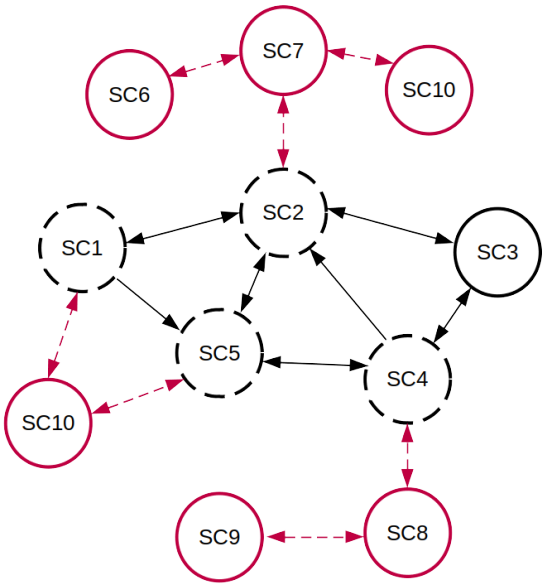

Figure 4.4. Dynamic Software Architecture - Case 2 [57]

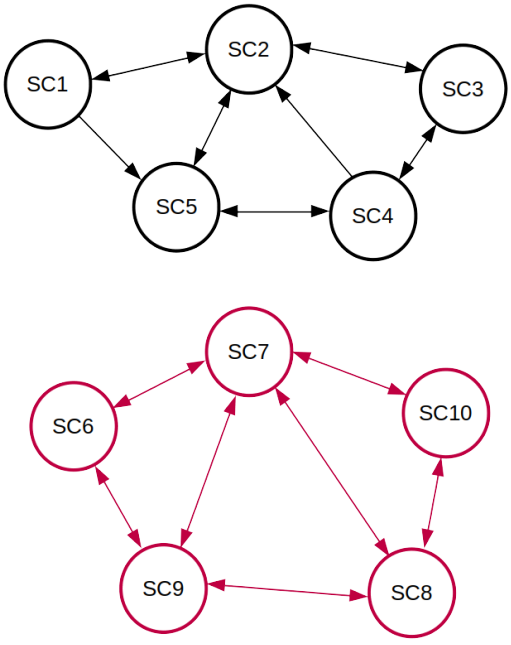

Figure 4.5. Dynamic Software Architecture - Case 3 [57]

on a controller attack or some other implementation or a corner case problem. Thus the system defaults to a simplified solution with reduced functionality but with increased safety. In such situations, a switching component detects the anomaly, and to stabilize the system, it switches the control to a secure controller component, which is much simpler and easier to predict and verify.

The smaller and more trusted controller can be formally verified much easier because it tackles a much smaller task. Its size allows it to live in a different memory space in the kernel or in a hypervisor making the solution much more resistant to attacks. The same rigor applies to the switching component. The secure controller and the switching component are considered critical and trustworthy, while the other blocks are not. That is why they are shown in a different color in figure 4.6. The gray boxes show the untrusted components, and the boxes with other colors show the trusted ones.

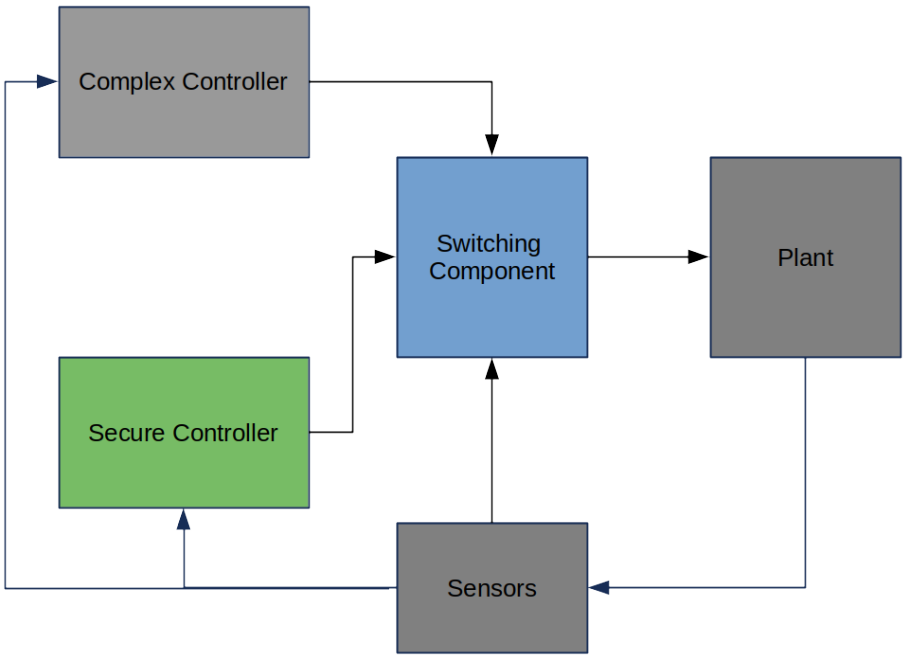

Figure 4.6. Simplex Architecture [60]

The development of the idea of the simplex architecture can lead to more elaborate schemes, as shown in figure 4.7. Here we introduce the concept of enforcers, which are

small and dedicated blocks that focus on certain concerns of the overall architecture. The approach is similar to the aspects in Aspect Oriented Programming (ASP) , where one aspect is universally introduced to be applicable in the entire system. We show a general control diagram for a robotic vehicle that has a high-level mission application, a controller, and an observer. A smaller controller called enforcer takes care of one concern that normally the standard controller handles. If this concern is violated, then the MUX switches the control to the enforcer, which starts controlling the system and thus keeps the system safe. The difference between the simplex architecture and using enforcers is that multiple enforcers could take care of different issues when needed. The approach is modular and flexible. It allows the designer to focus on a particular aspect and develop a guarantee for it without modifying the existing software. Enforcers can be added incrementally throughout the life of a system as new needs arise. Enforcers can focus on issues such as timing, logic, security, and other issues.

As shown in figure 4.7 the enforcer and the MUX or Trusted Execution Element (TEE) are shown in a different color since they are the ones that are trusted. The assumption is that they can be developed and proven to deliver expected behavior since their functionality is simpler and very focused. The other components are much more complex and unpredictable and impossible to verify with formal methods because of their size and complexity. One advantage of using enforcers is that they can be applied before or after the design and when the system evolves based on new concerns or even based on some that were missed during the design.

The following mathematical representation can describe the enforcement function E [60]:

$$E(I_{\mathrm{i}}, S_{\mathrm{i}}, O_{\mathrm{i}}) \rightarrow O_{\mathrm{i}}^{'} \tag{4.2}$$

where I_{i} are the inputs S_{i} are the states of the components

O_{i} are the normal outputs $O_{\mathrm{i}}^{'}$ are the outputs after enforcement

When the enforcer takes control then, the normal output is overriden by the enforcer's output:

$$O_{\mathrm{i}} = O_{\mathrm{i}}^{'}$$

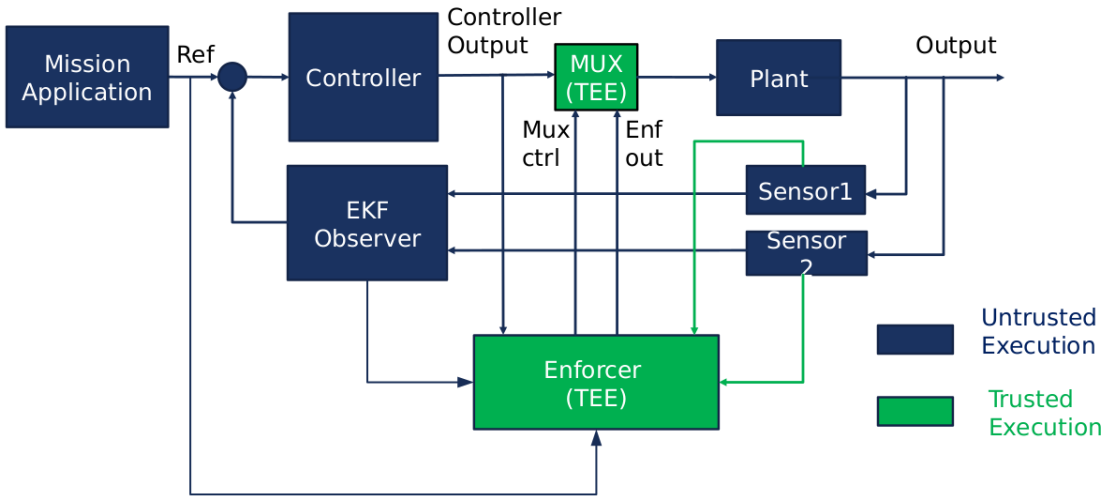

Figure 4.7. Run-Time Enforcers [60]

Another example of a dynamic approach is the software rejuvenation technique. The idea behind software rejuvenation was discussed in previous chapters. A typical implementation is shown in figure 4.8. If we assume that we have a control process that is running and we want to create checkpoints of its state periodically, then we can use these saved states of the process to perform restores periodically. An implementation can achieve better security if the stored image is in a protected location, for example, in the kernel, while the control process is running in user space. The scheme is a proactive way to guard against attacks against the control process. There are different levels of rejuvenation that can happen. One is to restart the entire OS if time permits. This approach may not work for the majority of OSes since the boot time is significantly large compared to the dynamics of the control object. On the other hand, restoring a process or a thread as part of a running process is

entirely doable for many applications. This is another example of dynamic architecture that can counteract intentional attacks against the controller, sensors, or actuators. The premise is that the reboots happen fast enough so that even if attacks happen between a checkpoint and a reboot, the state of the process can be recovered.

There are numerous aspects to consider when implementing software rejuvenation for CPS, especially for real-time systems. One is the memory requirements, as we need to save an image of the running application in memory and then restore it. The other important thing to consider is the time it takes to store the image when performing a checkpoint and the time to restore it. Since the application will essentially be frozen during this time, albeit briefly, we need to ensure that the system's overall dynamics do not get affected by the perturbation. The last point to consider is how frequently to perform a recurring rejuvenation. If performed too frequently, the controller may lose stability and crash or become sufficiently unreliable. Overall, even if the technique can be applied in many cases, some systems can be more sensitive than others. In such situations, some adaptation of the approach may be necessary, or it can even be impossible to use it.

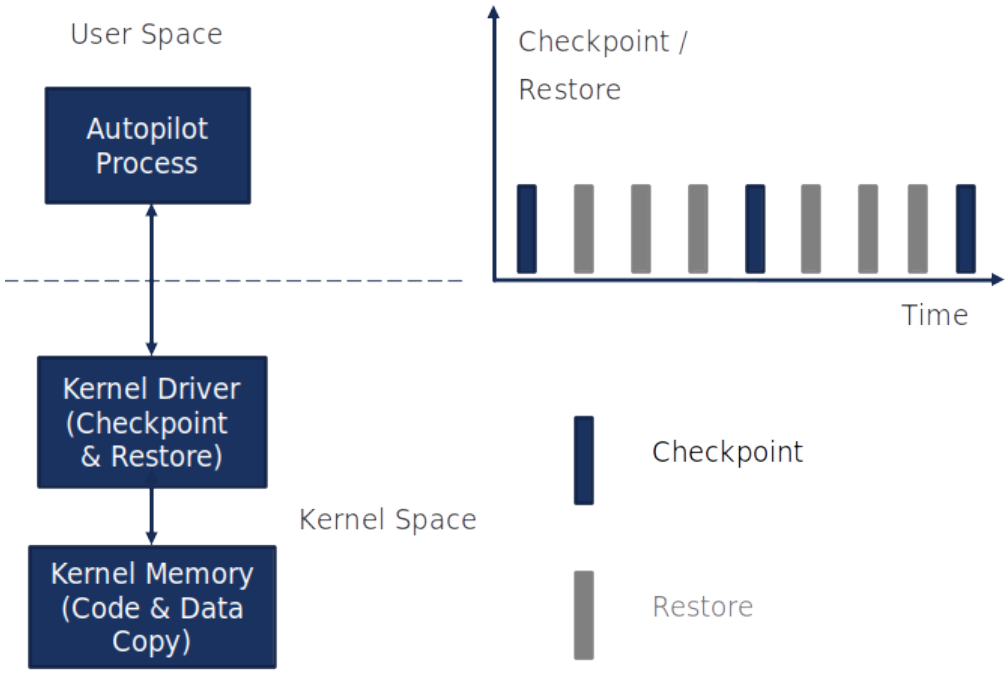

Figure 4.8. Rejuvenation of a running process [60]

4.4 Complexity Assessment of Dynamic Software Architectures

Software architectures are very different, and it is hard to think of their complexity without having a particular methodology. This is even harder for dynamic architecture, where the complexity can change as the architecture changes at run-time. Comparing one setup of software components with another can help predict the complexity of the architectural changes that are happening due to dynamic reconfiguration of the system. One of the challenges in calculating complexity is that the method used needs to be suitable for different architectures and it is possible to assess it quantitatively so that the architecturs can be compared objectively. This does not seem feasible across the board but is definitely doable for architectures following a common paradigm.

The complexity of different engineering systems can be studied by considering the components used in the system and the connections between them. This is true for software and mechanical systems. The goal of estimating complexity is to develop a quantitative mechanism that is applicable to different systems. Such a study is presented in [61]. The complexity is attributed to the complexity of the used components, their interfaces and the topology of the connections. This treatment is based on graph theory and utilizes the adjacency matrix of a graph representing the system under investigation.

Many systems today rely on some form of middleware to communicate between subsystems and components. The use of the Component and Communication (C & C) paradigm is also fairly popular in the software industry, where component reuse is increasing. Unfortunately, standardization of architectural approaches is hardly achieved; thus, many architectural styles exist. The three most widespread architectural styles are centralized, service-based, and publish-subscribe [62]. Service-oriented architectures are used for system of systems (SoS) and web-based deployments. On the other hand, Publish-subscribe is popular for robotic systems and is used in several popular open-source projects.

4.4.1 Publish-Subscribe Architectures and ROS for Dynamic Management

As described in many sources, ROS has been built with modularity in mind and is based on the publish-subscribe paradigm. Messages are handled in topics that a node can publish

or subscribe to. Every node can be started or stopped dynamically and can subscribe to and publish different topics. Since nodes can be implemented in different languages, such as C++ and Python, and can run on different computers, this allows for much flexibility in this architecture. Therefore ROS is gaining popularity in the research field as a good platform for experimentation.

ROS provides not only messages but services and actions in its ROS2 version, although services are based on messages and so are actions. The best attribute of the architecture is that all communications are asynchronous. This allows for nodes to be completely independent. The replacement of nodes becomes fairly easy as long as they provide the same messages, services, and actions processing of the component they replace. This also means that one component can be replaced by two or more components collectively providing the same interface.

4.4.2 Complexity Derivation for Publish-Subscribe Systems

One can expect that messages require the least processing and are completely asynchronous. Services, on the other hand, are typically synchronous and require a bit more processing. Actions introduce updates in addition to services and have the highest overhead. The functionality that happens once a message or a service is received can be arbitrarily complex and needs to be assessed on a case-by-case basis. This brings the complexity of algorithms, which is a well-studied area.

A complete methodology for deriving the complexity of publish-subscribe systems is provided in [59]. Equation 4.3 provides the formula for calculating complexity of component based systems. It is based on the complete interface definition of components. The architectural complexity is the sum of all components' complexities plus the number of subscribers per each published message. The main contributors to the complexity are the components and the work that they do. The message dispatching overhead is minimal and can be ignored in some situations.

$$AC = \sum_{\mathrm{i}=1}^{K} CC_{\mathrm{i}} + \sum_{\mathrm{i}=1}^{M} M_{\mathrm{i}} \tag{4.3}$$

, where $\sum_{i=1}^{K} CC_i$ is the most significant term formed by the summation of all components' complexities.

The term $\sum_{i=1}^{M} M_i$ accounts for the number of subscribers per each published message in the system and, in this sense, measure the interconnections between the components.

4.5 Conclusion

This chapter was devoted to different solutions for achieving system resilience. Some of the approaches are based on existing techniques, and some introduce new aspects of completely dynamic systems. The difficulty in utilizing such schemes is to validate their properties and to provide guarantees for the system in addition to increasing resilience. This would require run-time verification and validation techniques and modeling of system properties at design-time. Resilient systems have gained popularity in the last two decades and are becoming even more important as systems' complexity increases. Only some techniques that exist can be applied to every type of system. Thus a careful analysis of the run-time effect is absolutely needed. Resilience for system of systems (SoS) is even more challenging, as some of the next chapters explore.

5. DYNAMIC RUN-TIME BEHAVIOR FOR IMPROVING SECURITY

Security is an essential topic with many implications in different industries. For robotic systems, security can mean a direct assault on safety . This is true for all safety-critical systems, as they can affect people and infrastructure with costly consequences. There are significant differences between IT systems and safety-critical systems when security is concerned. With large IT systems, intrusion detection systems, firewalls, and security auditing scripts exist. With CPS these luxuries are too expensive or too slow to deploy or can't even be adapted to the memory footprint of some devices. Another significant difference is that safety-critical systems have physical sensors and actuators and interact with the environment in many different ways. This requires a different approach to security.

The approach that is more appropriate for CPS is based on dynamic security, which is a relatively new field of study. The idea is that all vulnerabilities cannot be known in advance and that intrusion detection cannot always be successful and timely. The detection of attacks is costly, and it may not always work. The number of attacks and their traits is vast and a moving target with the sophistication of the attackers, so the approach has certain limitations, even in IT systems. This leads us towards creating a dynamic environment so that the attacker has much more difficulty in understanding what is going on and therefore attacking the system [63]. The so-called Moving Target Defense (MTD) is a strategy that aims to increase the cost and difficulty of the attack. Although dynamic security is a new field, it is becoming relevant for IoT and other CPS as a proactive scheme that can thwart attacks from different classes.

The implemented dynamic aspects that can be used for improving security can be very different. They can fall into three major categories [63]. First, the selection of different architectural configurations is one way to provide dynamic changes. A second way is to constantly modify the behavior of the system. Finally, the timing of making all these changes can be varied so that additional randomness is introduced. All three categories can be used together so that the level of difficulty is increased and the system presents a significant challenge to a potential attacker . The time component is essential as any attacker needs

time to detect patterns and devise a strategy. If the time of steady behavior is too short, they cannot react and come up with a successful attack as the system changes constantly.

5.1 The Case for Security through Dynamic Component Management

One specific feature of safety-critical embedded systems is that they may not have significant computational power. Sometimes this is determined by the cost and role of their applications. For example, smart sensors that detect vibrations in a bridge structure may be deployed in large numbers and may run without batteries and generate electricity from the bridge's vibrations. They have to run a very small CPU with minimal consumption. Even if newer devices are coming with more powerful processors, many still use cheaper and smaller chips. In addition, stand-alone devices do not have the luxury of experienced IT staff that can mitigate the effects of a security breach. All these differences lead to unique approaches. One of these approaches is the preventive dynamic component management discussed in this chapter. The approach builds on the fact that software components and entire applications can be dynamically restarted periodically, creating a more unpredictable memory presence of the application.

This approach for security is fairly proactive and focuses on prevention rather than on detection and mitigation, as most other typical approaches. The advantage of a proactive approach is that it does not focus on the myriad possible threats. However, it just hardens the devices to withstand attacks, independently of their origin and mechanism. It uses time and space as changes can happen randomly in time, and software can be loaded to randomly different base memory addresses. A further improvement can be to use diversity when restarting components or applications by loading functionally equivalent but differently implemented components or applications. This dramatically changes the software's landscape and thwarts attack plans based on knowledge of memory maps and specific architectural patterns.

The concept of dynamic refresh of a software component is shown in figure 5.1. In the figure, a higher-level component manager is envisioned. It takes care of loading, unloading, stopping, and restarting one component with another; for example, A is replaced by B.

There are many prerequisites for this operation to be successful at run-time, especially when real-time systems are concerned. Some of the challenges come from a lack of standards and the use of COTS components from different vendors. These challenges will be discussed in the remaining parts of the chapter.

Replacing running components with functionally equivalent ones is a way to create dynamic changes that can lead to changes in structure, behavior, and time fluctuations. The system will still maintain its overall functionality. It simply morphs into new behavior and alternates it periodically while delivering similar quality of service and functionality while improving security. To do this, not just one but many components can dynamically be swapped with their replacements. Every time a component is loaded, it goes to a new memory address which is typically dynamically created by the operating system. Since it lives for a short time, its patterns are hard to detect as a new component is periodically loaded at a different starting address. As new components emerge and go away, the topology is restructured continuously. Therefore the technique is fairly good against memory attacks that rely on previously known memory layout.

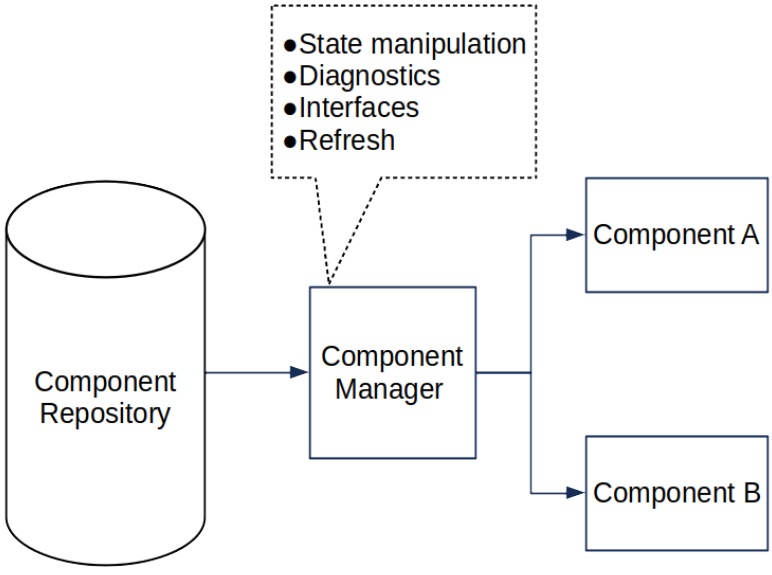

Figure 5.1. Dynamic Component Update [51]

5.2 Attack Model

The attack model that is envisioned for the dynamic management of software architectures is not a usual one. The usual attack model mentions particular vulnerabilities and ways to penetrate the system. This assumes the presence of a detection mechanism and a well-defined signature of each attack. The standard attack model definition also assumes a particular response or mitigation against each attack . In this case, there is no assumption on the types of attacks. There is no known attack signature that the application needs to store in a database. There is no particular detection mechanism either.

The prevention scheme considered here is not concerned with the way the attack is initiated as it tries to make the attack impossible by creating no-static conditions that significantly reduce the risk or even eliminate it in some cases. The only assumption that is considered is that the attacker has a way to get to the memory of the running system and can modify its contents. This can allow the attacker to modify control algorithm coefficients or sensor and actuator variables to affect the control software's normal operation. Such memory attacks can happen through buffer overflow or other techniques, as described in he literature review in chapter 2.

Since the dynamic security approach is designed to guard against memory attacks targeting the control application, the attack model can utilize any method to start this attack . One such attack scheme is through the communication channels and can be using the buffer overflow vulnerability. Another method if the CPS uses a Linux-based implementation is to launch attacks through the /proc file system. Either of the methods relies on knowledge of the controller's memory layout and the program's starting address in memory. Other similar attacks also rely on static information that is well-known in advance based on the characteristics of the system. For example, the attacker may know the version of the autopilot system and have the source code for it, thus being able to exploit the memory layout of the running instance of the autopilot software.

5.3 Specifics of Dynamic Mechanisms for Security

As shown in figure 5.1, the component manager needs to consider many different aspects of each component in order for the scheme to work. These are discussed in detail in a previous work [51]. The main issue to consider is to provide well-structured interfaces for components as shown in the previous chapter, so that different vendors can deliver compatible solutions. The specific new additions to the interfaces that are considered critical here are the support for diagnostics and state manipulation of the component. The diagnostics and control portion of the interface allows for the component's starting, stopping, and loading and unloading. Extracting the current state allows for transferring the state to a newly loaded and compatible component that can continue to function without perturbing the system. As shown in figure 5.2 we can see that each component starts by being created and then can move to a running state. From there, the component can be stopped, enter the blocked state, and be unloaded, destroying its instance in memory. The same state diagram can be used for subsystems or even entire systems.

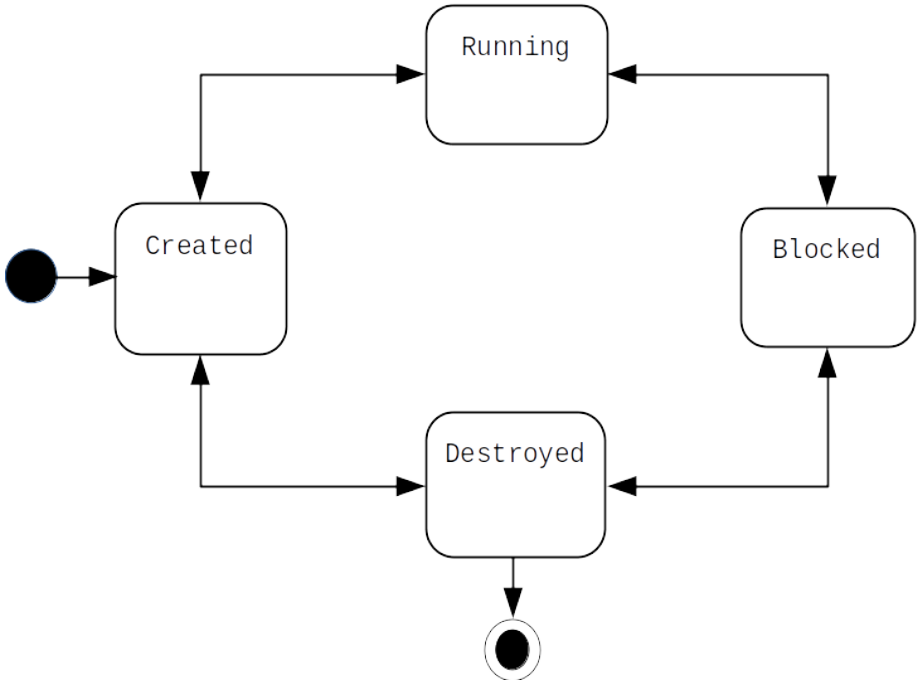

Figure 5.2. Component States

The actual exchange of two equivalent components is shown in the sequence diagram in figure 5.3. The diagram shows all the steps to perform the exchange without significantly affecting the system. Stopping a component may take some time as the component needs to finish its current transaction before a new component can run. The operation assumes that the two components are equivalent in their interface and internal state and provide the diagnostics and control mechanisms to start, stop, load, and unload them. The diagram can be elaborated for cases where more than one components replace a single component, but the mechanism is similar. The sequence assumes that there is no global state and that each component depends on its own state, which is independent of other components. This assumption is certainly valid for a publish-subscribe architecture and others that are designed with the principle of loose coupling.

Another important issue that needs to be considered is protecting sensitive parts of the system, like the component manager. Since we assume that the component manager is working to protect the software components by continuously restarting them, it is paramount to think about how to protect it. To do this, we can use a compartmentalization technique that puts sensitive parts of the system in areas that cannot be affected by the other parts of the system. This can be allowed through hardware memory protection support from modern chips. This is also a prerequisite for a system software solution that takes advantage of these hardware features, such as OS or a hypervisor.

Figure 5.4 shows the placement of the component manager in kernel space. This makes attacks that target user ineffective to the component manager . The assumption that it resides in the kernel is just an example as it may exist in another secure place such as a hypervisor too. The important fact is that it is not part of the user space where the application resides. The principle that is followed here is that we need to rely on some form of a Trust Execution Environment (TEE) that guarantees privileged access to some regions of memory. This separation technique is not new, but the implementation of the approach in the light of dynamic software management systems is new.

One side effect of using dynamic component management is the actual demand for more resources, such as more memory and higher CPU load. Therefore it can be applied judiciously to a system taking into consideration the specifics. Every system has several important

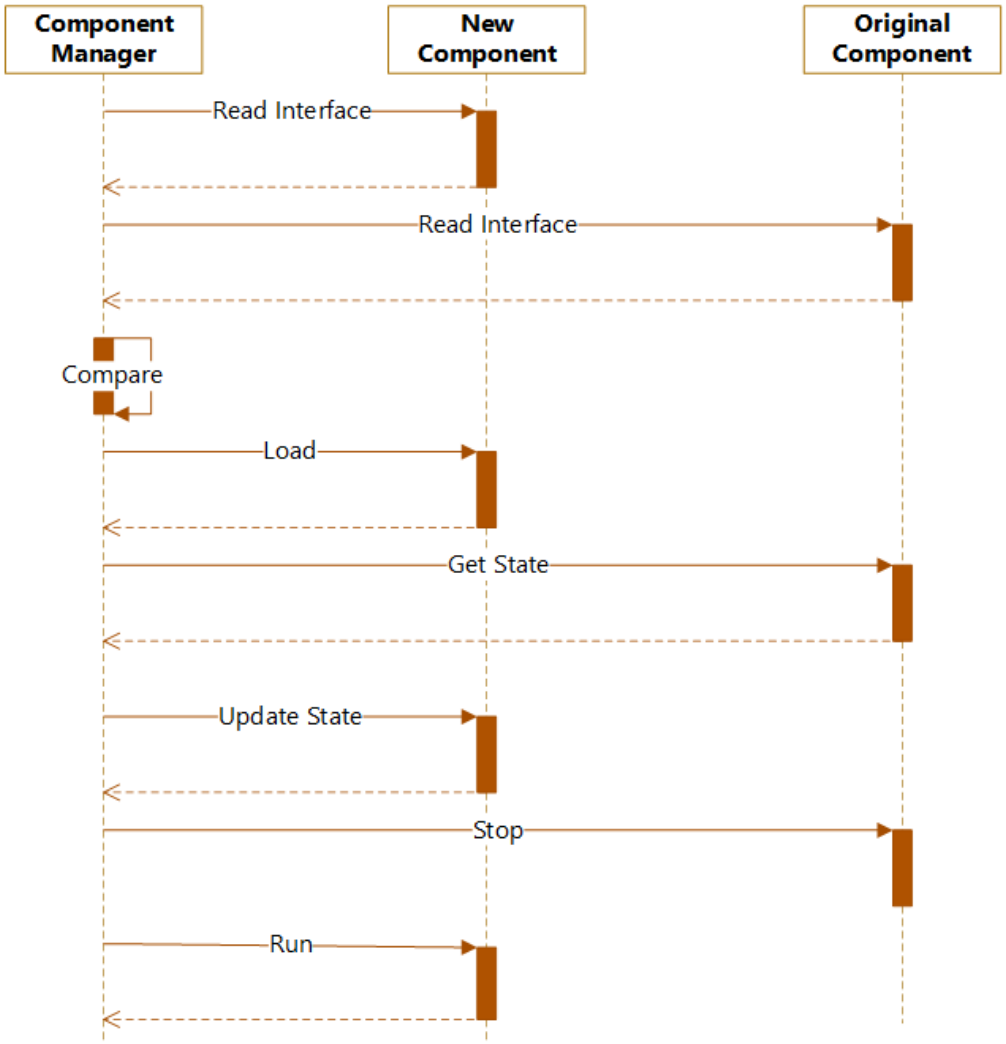

Figure 5.3. Component Exchange Activity Diagram [51]

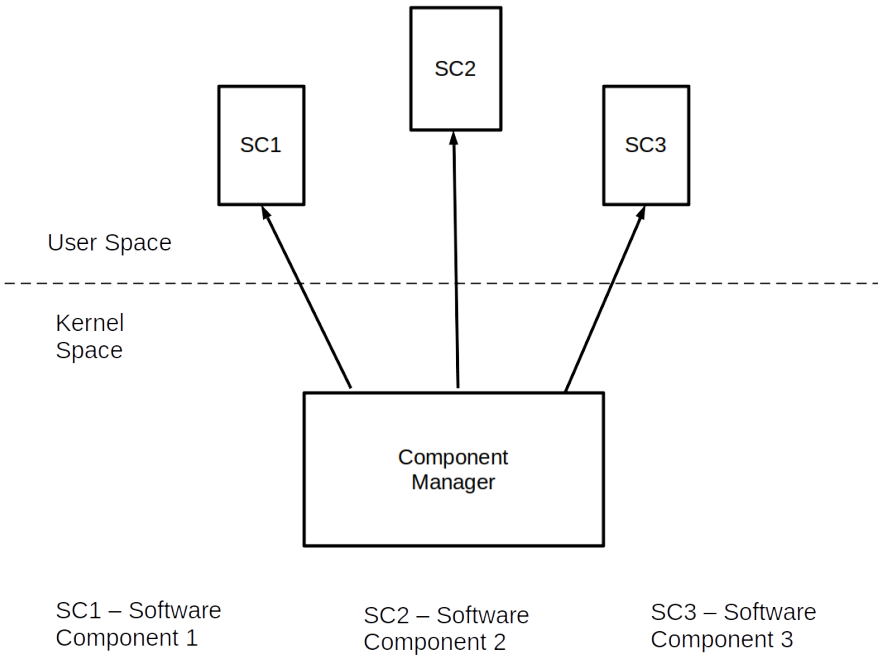

Figure 5.4. Kernel Implementation of Component Manager [51]

components that are the most important from safety and security standpoint and need to be guarded the most. Applying dynamic management only to them is a good approach that can spare some computational power compared to when applied to all components. The other factor that can be used judiciously is the time concerning how often components are replaced. This decision can be made based on how much time a potential threat takes to be realized. If a threat can be realized in seconds, then component replacement does not need to happen very frequently to be effective. This also affects the overall increase in the consumption of computational resources.

Testing the effects on the overall mission is a high-level approach that ensures the mission is still possible, and the mission quality does not suffer below a certain level. Any dynamic changes need to be assessed so that the system's overall operation is not negatively affected. Such a study on a UAV is done in [51]. Some authors refer to this as mission assurance, as discussed in later chapters of this work. In addition, an analysis of the effects on the system's power consumption is important as higher CPU loads and memory usage leads to increased

power consumption which can affect battery life in CPS. This can encourage us to use the technique only when we consider that the system operates in a hostile environment or when a threat is detected. This assumes some form of adaptation ability concerning security.

5.4 Case Study

To experiment with these techniques, we chose to use the PX4 autopilot software, an open-source platform. The PX4 is predominantly used for UAVs and supports simulation platforms such as Jmavsim and Gazebo . The mission of the UAV was controlled through MAVSDK , which is another open-source platform. The idea behind the experiments was to continuously restart a key software component, such as the position controller or the attitude controller, and to monitor the ability of the system to execute the mission. The mission was selected to be several minutes long and challenging in order to assess the effects of the dynamic changes in flight. The experiments described in this work are similar to this previous work, although here we provide more details [51].

The component that is constantly being replaced is the position controller in PX4. The original component is in the mc_pos_control folder. A separate folder is created to host the new implementation of the position controller, called mc_pos_control_backup. The state is saved and restored through serialization techniques, using an open-source C++ library called bitsery. The rate and attitude controllers have also been used in the experiments, although the position controller is a more compelling example. The component manager implementation is shown in listing.5.1

Starting the PX4 autopilot can happen through the following sequence that starts the component_manager, and after the mission is run, the component_manager can be stopped. This px4 command sequence is shown in listing 5.2. These commands can be entered at the PX4 command shell after it starts.

The mission plan needs to be created by the QGroundcontrol application and saved as a separate file, for example, test_flight.plan. To start the mission application MAVSDK needs to be installed, and the following commands need to be executed as shown in listing 5.3. The example shows a MAVSDK application that uses a flight plan in the form of a JSON

Listing 5.1. Component Manager Main Function

```
int component_manager_thread(int argc, char *argv[])
{
        PX4_INFO("Component Manager!");
        bool alternate = true;

        while(!thread_should_exit){
                if(alternate){
                        px4_daemon::Pxh::process_line
                          ("mc_pos_control set_state",true);
                        PX4_INFO("Starting mc_pos_control_backup!");
                        usleep(300000);
                        px4_daemon::Pxh::process_line
                          ("mc_pos_control_backup start",true);
                        PX4_INFO("Stopping mc_pos_control!");
                        usleep(300000);
                        px4_daemon::Pxh::process_line
                          ("mc_pos_control stop",true);
                        alternate = false;
                }
                else{
                        px4_daemon::Pxh::process_line
                          ("mc_pos_control_backup set_state",true);
                        PX4_INFO("Starting mc_pos_control!");
                        usleep(300000);
                        px4_daemon::Pxh::process_line
                          ("mc_pos_control start",true);
                        PX4_INFO("Stopping mc_pos_control_backup!");
                        usleep(300000);
                        px4_daemon::Pxh::process_line
                          ("mc_pos_control_backup stop",true);
                        alternate = true;
                }
                usleep(300000); // 0.3 Sec

        }

        thread_running = false;

        PX4_INFO("Component manager Exiting");

        return 0;
}
```

Listing 5.2. PX4 Commands

```
make px4_sitl jmavsim
pxh> component_manager start
... Run test
pxh> component_manager stop
```

Listing 5.3. Mission Application

```
cd ~/MAVSDK/examples/fly_qgc_mission
mkdir build
cd build
cmake ..
make
./fly_qgc_mission udp://:14540
test_flight.plan
```

file created with the QGroundcontrol application. Since the mission executes for minutes, the system can go through hundreds of replacements of the position controller component.

The saving and restoring of the state of the components happen through the following two functions in the two components, the original and the alternate component - listing 5.4. Some representative serialization and deserialization routines are shown in listing 5.5. The entire implementation is not shown in the interest of saving space. The approach can be used for components with different complexity and implementations other than C++. Serialization is even supported naturally in languages such as Java and readily available to C# and Python.

The same mission application was run in the normal case without any changes in the code. The application was also run in the case where the component manager ran, and component replacement happened continuously, as shown in listing 5.1. We can refer to this case as an alternate case. The following plots show the comparison of key parameters of the normal case versus the alternate case. We can see that there is some deterioration in all plots, but overall, the mission was successful, and the vehicle did not visibly show any issues in the simulator. The test was fairly aggressively performing operations faster than a second, which gives a lot of protection against an attacker who needs time to analyze the environment.

Figures 5.5 and 5.6 show how the trajectory has been affected by the dynamic behavior. Figures 5.7 and 5.8 show the position X. The remaining figures show the effects of the restart of the position controller on the system. Since we preserve the state before loading a new

Listing 5.4. Serialization/Deserialization of Position Controller

```
bool MulticopterPositionControl::set_state(){
        start_mc_pos_serialization();
        ser_takeoff_state(_takeoff.getTakeoffState());
        ser_vehicle_local_position_setpoint( &_setpoint);
        ser_vehicle_control_mode( &_vehicle_control_mode);
        ser_timestamp_last_loop(_time_stamp_last_loop);
        stop_mc_pos_serialization();
        return _control.set_state();
}


bool MulticopterPositionControl::get_state(){
        start_mc_pos_deserialization();
        TakeoffState t_state;
        deser_takeoff_state(&t_state);
        _takeoff.setTakeoffState(t_state);
        deser_vehicle_local_position_setpoint(&_setpoint);
        deser_vehicle_control_mode(&_vehicle_control_mode);
        deser_timestamp_last_loop(&_time_stamp_last_loop);
        stop_mc_pos_deserialization();
        return _control.get_state();
}
```

Listing 5.5. Serialization/Deserialization Routines

```
void start_mc_pos_serialization(){
        s_ptr = new std::fstream {mc_pos_state_file_name,
                s_ptr->binary | s_ptr->trunc | s_ptr->out};
        ser_ptr = new bitsery::Serializer
        <bitsery::OutputBufferedStreamAdapter> (*s_ptr);
}

void stop_mc_pos_serialization(){
        ser_ptr->adapter().flush();
        s_ptr->close();
        delete(ser_ptr);
        delete(s_ptr);
}

/* Deserialization routines */
void start_mc_pos_deserialization(){
        s_ptr = new std::fstream {mc_pos_state_file_name,
                s_ptr->binary | s_ptr->in};
        des_ptr = new bitsery::Deserializer
        <bitsery::InputStreamAdapter>(*s_ptr);
}

void stop_mc_pos_deserialization(){
        delete(des_ptr);
        delete(s_ptr);
}
```

position controller instance, we can see relatively small perturbations. The slight degradation in flight quality is definitely better than a failed mission in real life, especially in the case of safety-critical systems. The main goal is to increase security while still accomplishing the mission.

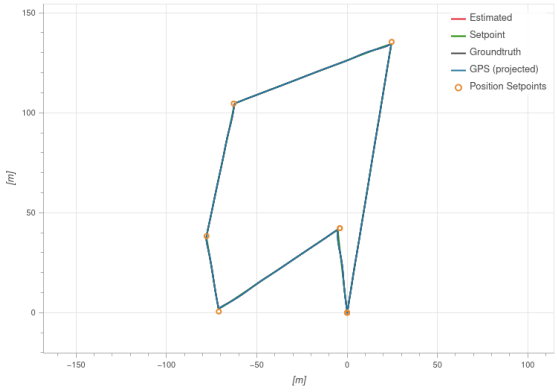

Figure 5.5. Trajectory Normal Case

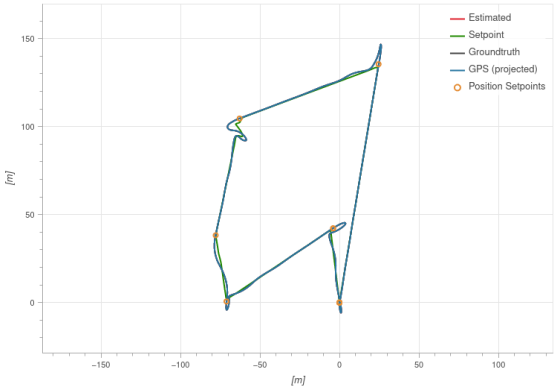

Figure 5.6. Trajectory Alternate Case

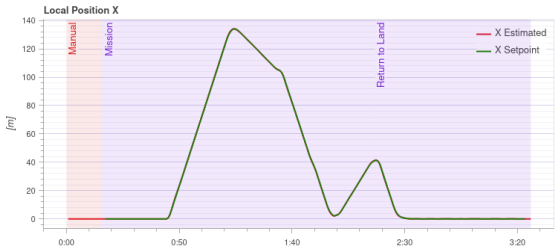

Figure 5.7. Local Position X Normal Case

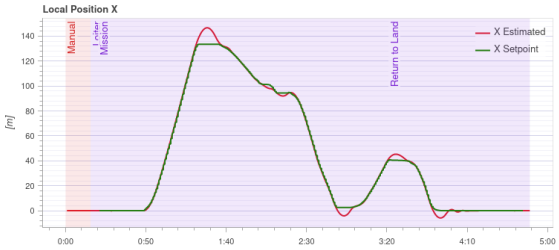

Figure 5.8. Local Position X Alternate Case

5.5 Conclusion

Dynamic changes during run-time are a novel method to counteract security threats. We discussed the preconditions for devising such a scheme and the potential advantages and drawbacks. It needs to be fully studied and its application domain is particularly suited to smaller embedded devices that do not have traditional fortification tools like large IT systems. The approach can affect how systems are designed and deployed and the potential adoption of standards for software components and their interactions. The case study proved that

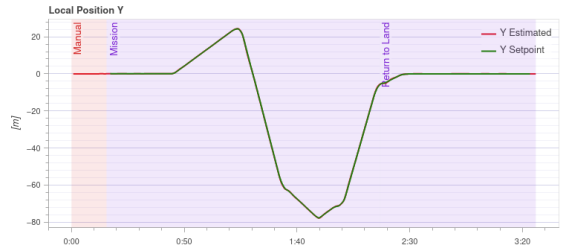

Figure 5.9. Local Position Y Normal Case

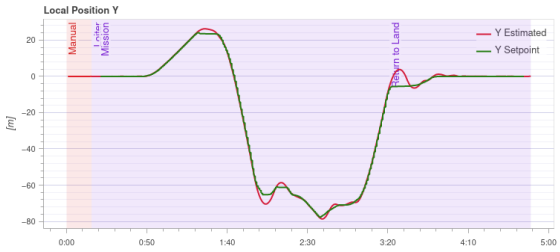

Figure 5.10. Local Position Y Alternate Case

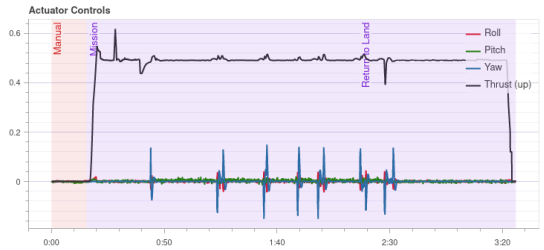

Figure 5.11. Actuator Controls Normal Case

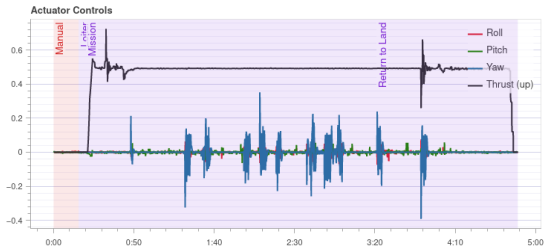

Figure 5.12. Actuator Controls Alternate Case

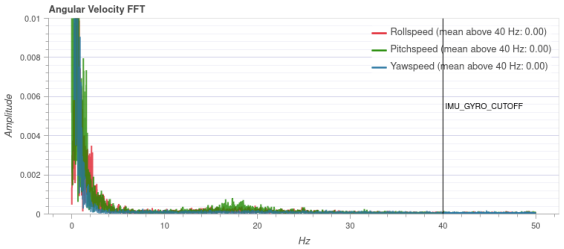

Figure 5.13. Angular Velocity FFT Normal Case

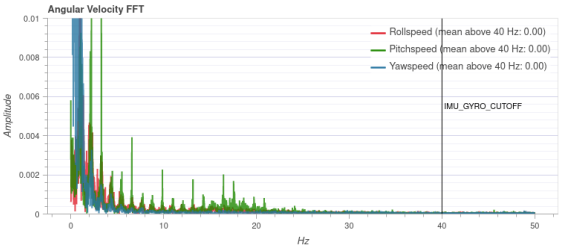

Figure 5.14. Angular Velocity FFT ALternate Case

the approach is feasible for robotic vehicles such as UAVs. Since the case study was based on a world-class autopilot as PX4 with significant complexity, this guarantees that smaller systems can do even better with the approach. The discussion here may have touched only on some of the possibilities but also helped lay a foundation for future explorations and experiments.

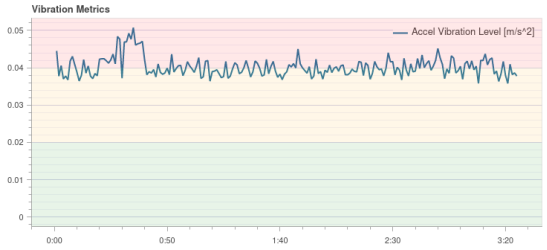

Figure 5.15. Vibrations Normal Case

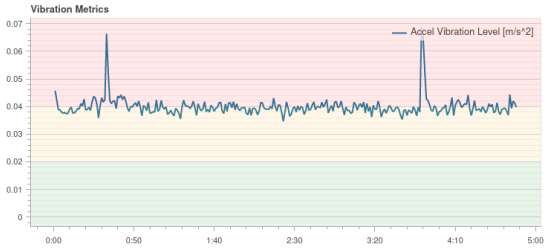

Figure 5.16. Vibrations Alternate Case

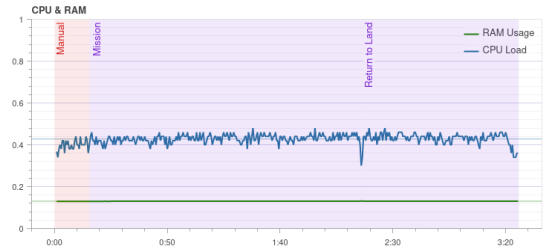

Figure 5.17. CPU & RAM Normal Case

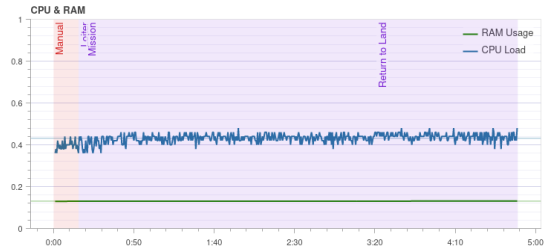

Figure 5.18. CPU & RAM Alternate Case

6. MODELING TECHNIQUES FOR DYNAMIC ARCHITECTURES

Any model is a high-level system abstraction, emphasizing only some aspects of the system. Therefore models may not be considered the ultimate truth by many. They could be viewed as a mechanism to help understand the system and easily create the right architecture without the final details. In some cases, languages like UML have been used to help exchange ideas and create documentation. In many cases, models tend to be visual, although, in some fields, textual representation has long ago set permanent presence, for example, VHDL for representing hardware designs. Higher-level text-based programming languages provide a mechanism for modeling while writing code [64]. This eliminates the need to have a modeling language, an implementation language, and the necessary transformations between the two. We have many modeling and programming languages in the software domain. Many tools to do transformations between them exist. The desire to provide a language that can handle all levels of abstraction and implementation is hard to achieve. Human perception also plays a role in the proliferation of visual languages such as UML and SysML . It is preferred as a design method, documentation, and modeling approach to help convey ideas between different parties.

Modeling can be a money-saving approach and, most importantly, a safety guarantee for complex systems. Unfortunately, many modeling platforms present a partial methodology for designing a system. It has been proven that models could have avoided significant design disasters that were not proactively found during the design phase [65]. One of the achievements in the last decades has been the standardization of modeling languages. Another direction has been the generation of executable code from models such as AADL, UML, SysML, Matlab , and others. The successful platforms have been the ones that are standardized or supported by large organizations. Another factor for success has been the generality of modeling languages. Some languages like AADL and SysML allow for the creation of extensions that can bring domain-specific features that can be extremely valuable for the domain experts involved with modeling. In AADL, they are called annexes and allow

a lot of flexibility but require significant time and knowledge to make them a success in the real world.

Formal modeling requires significant investment in better tools and processes to succeed. A large percentage of organizations use modeling informally [65] to address issues such as concept understanding or documentation of the architecture. The use of models in dynamic systems can happen through strict coordination between the models and implementation, and it must continue through all development phases. The argument about whether the truth is in the model or in the code may be controversial, but the final product needs to have code that reflects the model and model that reflects the implementation. The cohesion between the different aspects of the design artifacts, models, and code is a major goal that has not been completely and universally resolved.

6.1 Modeling Techniques

There are various ways to devise a software architecture for a system. Traditional design methodologies have used requirements, design and testing documents for all phases of the development a new system. Model-Based Software Engineering (MBSE) has so many benefits compared to traditional methods, such as documenting the design, developing a proof-of-concept, maintaining the model as the system evolves, and others. When the system changes, the model can track the system changes, and verification can be performed on-the-fly assuring the system parameters. The key to such model effectiveness is to create a bidirectional link between the model and the implementation. Modeling and simulation are possibly the cheapest and safest way to prevent disasters in the real system, and that is why modeling has been getting more attention recently.

The typical flow when modeling is done is to use the top-down approach. This seems to be favored by seasoned modelers as the approach allows for a cohesive design that links the components into a solid architecture [66]. When a bottom-up approach is used, then the modeler can use proven solutions in the design as low-level kernels and other verified modules [67]. The reality is that both techniques complement each other, and even if the

top-down approach leads the overall design, the bottom-up techniques allow for the reuse of components that are created and verified by other techniques.

The biggest challenge in modeling systems that have dynamic behavior is to represent the changes of the components as they are expected to happen and to discover their effect on the system. Some of the most important analyses that need to happen are fault analysis and end-to-end time analysis . These two analyses are directly related to safety. The result of the modeling phase needs to determine that the dynamic changes do not introduce adverse effects such as new faults and that the timing is not affected according to the timing requirements for the system. This can be done if the software components present information on timing and state in their interfaces.

Another challenge when doing modeling for a real-world system is that one modeling language may not be sufficient to represent all characteristics of the modeled system. A combination of architecture models with AADL or SysML and models of physics and algorithms through platforms such as Simulink and Modelica is very likely to address the majority of the modeling needs for many stakeholders. This makes code generation, integration, verification, and validation very challenging and presents dependencies on many tools from different sources. Making all this work together is much more time-consuming, error-prone, and incomplete. A better approach is to have one modeling language that can address all concerns, and although the design of such language is fairly compelling, there are some recent attempts in this direction [68].

A scenario that is often followed in the creation of safety-critical systems is to use of modeling languages to address various aspects of the design. Such an approach can use SysML or UML to define the high-level design, Simulink for the algorithms, and maybe a Domain Specific Language (DSL) for defining the real-time requirements for the system [69]. A concern with using several very different tools is to make sure that these same tools will be available in the long run, as the dependencies of the design on tools need to be supported through the maintenance phase. The code generation from different modeling languages is also challenging as the final code needs to be assembled together and reviewed, which is a slow and expensive process. The ideal situation would be to use one modeling language for everything and to be able to generate code from it. This may become more realistic with

richer languages that do not focus only on the high-level system level architecture but allow for the development of algorithms and real-time allocation of threads and communication between them.

There are some challenges in using only SysML for modeling; otherwise, this modeling language is the best candidate to represent the system architecture. This capability extends to large-scale systems and system of systems. The latest version allows for SysML to be extended with Domain Specific Languages (DSL) based on the KerML language as specified by the Object Management Group (OMG) . This direction is sometimes a preferred direction, compared to creating a brand-new DSL. The community that is already used to dealing with a particular language can find it easier to add some minor extensions as opposed to learning a new language.

SysML, especially version 1, is considered to be a semi-formal modeling language that may not address all modeling challenges for systems with dynamic properties. Since there are elements of uncertainty when dynamic reconfigurations occur, we need some special tools. Known configurations can be addressed through SysML stereotypes as an extension to the language. For unknown configurations, though, some authors suggest Stochastic Process Algebra (SPA) [70]. This approach can add some probabilistic behavior that can be integrated into the analysis. A promising direction is to augment existing modeling languages with SPA to provide this analysis capability.

In order to represent a dynamic architecture in a modeling language, several characteristics of the language need to be present. First, the language needs to support dynamic component instantiation. Second, dynamic interfaces need to be supported with new services being offered or removed from the interface. Finally, an event-driven architecture reconfiguration has to be possible. One such language that supports these mechanisms is EmbeddedMontiArc [71], which is a Component and Connector (C & C) language. The presented example in the reference shows a platoon of vehicles resolving a dynamic reconfiguration problem.

Many modeling languages support modes that can partially help in modeling dynamic architectures. One such modeling language is AADL because it supports modes in a particular architecture. Unfortunately, modes can only partially help and are not useful in truly dynamic systems where a large number of reconfigurations are possible. Therefore, the

modeling language needs to support service-based dynamic components, dynamic interfaces, and reconfiguration mechanisms. This requires either specialized modeling languages such as EmbeddedMontiArc or an extension of an existing one such as SysML. A preferred approach that is favored by many is to extend an existing and widely accepted language and provide the necessary tool for the modeling and simulation of dynamic systems. This may require a serious investment in time but has a better chance of success.

Many Architecture Description Languages (ADL) do not support dynamic behavior, which is a challenge when they are used in modeling modern systems. The π-ADL language is specifically designed to support dynamic architectures. It is based on higher-order π-calculus for the dynamic run-time properties and modal μ-calculus for the expression of behavior. In addition, it is considered a formal language which makes it suitable for modeling and code generation of dynamic systems [72]. It complements the many semi-formal languages, such as UML and SysML, that are used today but are not able to provide formal verification. Π-ADL also uses the Component & Connector (C&C) paradigm to define architecture as other languages do. Each component can have one or more ports that allow for connections to be established between different components. The architecture can be described through the ports' definition and the components' behavior. As the language is formal, this leads to the easy creation of tools and integration with other existing languages and tools. The creation of such tools contributes to the success of any language independently of its originality.

One aspect that needs to be considered when we want to use modeling is what would be the generated language for simulation. There are options to perform a model-to-model transformation to another modeling language such as Modelica [73] from the higher level system modeling language such as SysML or UML. Another option is to do a model-to-code generation, produce C, C++, or Rust code, and then run a system simulation. With SysML representing the high-level modeling view, a transformation to some executable environment is needed. The code generation approaches are discussed in more detail in the following section.

6.2 Code Generation of Systems Based on Models

Code generation is possible from many modeling languages such as AADL, SysML, UML, Matlab, Modelica, and others. Some of them have their own tools, like Ocarina for AADL [74], which allows code generation for C or Ada implementation languages. Matlab generates C code according to safety standards such as DO-178C and DO-331 [75]. The generated code complies to the Misra-C software standard. SysML also is supported by MagicDraw and IBM Rational Modeler to generate code [76]. Modelica has also been used to generate C and C++ code in certain domains [77]. This demonstrates that the model-to-code generation path is considered one of the standard ways to move from architecture to implementation.

An important issue that needs to be explored is ensuring that the model and the generated code have semantic consistency. This means that the generated code reflects the model and that the simulation of the model is in sync with the execution of the generated code [78]. Even if the model is validated if the generated code does not faithfully represent it, there will be inconsistencies in the implementation. It is essential to determine the data elements, events, states, and transitions in both domains and to verify that they have identical behavior. This analysis can be performed manually or with tools, provided that such tools exist.

One of the most popular code generation techniques from models is to use Hierarchical State Machines (HST) described in a model language such as SysML or UML. HSM allow for sub-states to exist, thus reducing redundancy in specifying the system [79]. Such schemes still require developers to add manual code after the code generation is done. The method helps minimize the errors that would be introduced if manual coding was used to implement complex state machines. Overall, state diagrams and their representation in a modeling language have a more straightforward mapping to code compared to sequence diagrams or activity diagrams, for example.

Code generation is a complex process that requires understanding of the model language, the implementation language, and some code generation techniques. The creation of a code generation tool can be very time-consuming. One approach advises using Machine Learning (ML) to gather the mapping rules the model to the implementation language based on examples [80]. The method uses Abstract Syntax Trees (AST) to map the two languages.

An alternative to the ML approach, which is data-driven, is to have a model of how the two languages map.

As code generation needs to translate constructs from the model to code, it can benefit from model-to-model transformation if necessary. Such transformation is discussed in [81] where a UML model is first translated to an ANSI-C model. A second phase takes the ANSI-C model and produces C code as a final output. The approach demonstrates that the actual language generation can be platform specific and can be relatively static for a particular platform. However, the model-to-model transformation can evolve as newer aspects of the modeling language are added to the code generation process.

Generating source code for any platform is too ambitious but focusing on a certain platform is much more realistic. This is true, especially when the platform defines a lot of boilerplate code for its architectural entities. One such platform is ROS, as it specifies the nodes and their interfaces in a fairly standardized way. Many works explore the generation of code from UML or SysML to ROS. One such work looks at the possibility of generating code from UML/MARTE to ROS [82]. The approach takes advantage of Aspect-Oriented Model Driven Engineering to address different concerns when modeling. The code generation uses a template-based approach representing the mapping from UML to ROS in XML files. The fact that each ROS node is a separate process with a main() function in C++ and a well-defined code structure significantly makes the task more practical.

Some authors suggest using DSLs for specifying the architecture and then generating code for a platform such as ROS. The approach aims to generate safe code in some works, and its correctness can be proven. The following work [83] suggests using the Coq language, thus achieving formal verification of the solution. This can help certify the code generation approach so that there is a guarantee of model-to-code conversion correctness. This code generation focuses on the boilerplate code used to structure the ROS implementation. The rest of the implementation that is done by hand needs to be verified separately.

There is a trend for programming languages to become higher level and, therefore, close to modeling. On the other hand, some modeling languages have support for textual syntax. These trends move the modeling languages closer to the implementation languages. This prompts us to consider how to merge the modeling and implementation into one with the

benefits of being able to do modeling in addition to having executable code. For this to become practical new types of tools are necessary. One example is the textual version of SysML, which has fairly low-level syntax for a system modeling language. This makes it a good candidate for describing the systems with enough precision to leave less for manual code additions.

The modeling languages have some differences compared to programming languages worth mentioning [84]. One such difference is that they can be visual and, therefore, not very formal. The semantics of a modeling language can be open to interpretation and, therefore, difficult to compile directly to an executable because of the abstractions from the operating environment. Their specs are also way more extensive and ambiguous than the grammar of programming languages. Their philosophy is on creating high-level concepts that can help the design and understanding of the system. Some of the latest textual-based modeling languages have better semantic structures. A trend toward formality in modeling languages can make code generation and simulation easier and propel modeling into a mainstream design approach with the possibility of generating high-quality production code.

The specifics of the modeling language and the implementation language are essential to the code generation process. If similar concepts can be easily mapped, the generation can be performed more straightforwardly. An example of such a good match is described when code generation is performed from π-ADL to the Go programming language [85]. The fact that both languages are based on π-calculus makes the mapping between components and channels straightforward. As the Go language implements goroutines and channels as embedded constructs, it can directly express the model in the implementation. Π-ADL is designed to support dynamic behavior in systems in addition to structural architecture, as most ADLs do. This makes it a perfect candidate for code generation for modern languages with built-in support for parallelism and support for building distributed networking applications.

6.3 Verification and Validation of System Models

The process of verification and validation strives to prove that the model correctly represents the requirements of the system and is generally correct and free of errors. Since the

model is part of the system design and implementation, the V & V process needs to start with it. Some authors suggest an organizational process that includes domain experts for validating models [86]. The best approaches require starting the process early and making it as automatic and formal as possible. This can be accomplished when the verification and validation rules are present in the modeling language and embedded in the model.

Other authors suggest creating a catalog of requirements for the models so that they can be checked as models are created. The requirements can be verified by domain experts or automatically. SysML models can be verified through scripts [87] written in different languages. This type of automation can be very useful in large models where human inspection can miss some subtle errors. It is best if the verification and validation is built into the modeling tool to guide the modeling engineer. This is another example of how modeling can be made more successful in the industry through the existence of proper tools with rich functionality and capabilities.

The complete verification and validation of a model may be impossible and too costly. In this case, a set of criteria can determine if the validation is good enough. Sometimes the cost of this activity may turn out to be excessive and impractical [88]. Whether the model is valid can be made by the team developing the model, by the users, or by an independent authority. The last choice is more expensive but provides more rigor and neutrality to the decision-making of whether the model is valid or not. Special focus needs to be exercised on whether the data used to generate the model is valid.

A critical aspect of validating systems is to start by validating their missions. In a complex system, the correct behavior is determined by system's capability to complete its mission. A mission can be considered as a sequence of actions [89]. A special mission language can even specify it. The mission elements can then be translated into executable code or formal models representing the mission. Each action can be described with pre-conditions and post-conditions as well as invariants. Proving that each action is doable can lead to validating the entire mission. This type of validation is independent of the architecture of the system or SoS.

The validation of the model can happen through a transformation to a different language. A typical modeling language such as SysML is not entirely formal, and some authors suggest a

transformation to a more formal language to validate the model. Such a language can be, for example, Event-B [90], a language used for system modeling and analysis. The bidirectional transformation from SysML and Event-B may be challenging, which is inconvenient. The approach can be used best with state machines expressed in SysML and then formally verified in Event-B. Other SysML abstractions, such as activity diagrams and sequence diagrams or block diagrams, may require different verification techniques that are not based on events.

Dynamic architecture can be verified through formal modeling languages such as Alloy . The dynamic extensions proposed in [91] allows for actions that can modify the states of components that are part of the architecture. Dynamic Alloy is developed based on the Alloy modeling and verification language. Alloy is based on first-order logic, has formal grammar, and has easily understood syntax and semantics familiar to software engineers. The dynamic architectures can be represented through graph grammar, which makes the approach high-level and universal as it is agnostic to specific architectural specifics. Dynamic reconfiguration can then be represented as production graph morphing rules describing possible transitions. These rules can be defined at design time and can be executed at run-time when the conditions for them arise. The model can be visualized and simulated through the Alloy Analyzer tool. Alloy's language allows for the creation of formulas that have pre-conditions and post-conditions. The user defines assertions that are automatically verified which is one of the outstanding features of the language. The main analysis in Alloy is called model finding, which allows for finding all possible dynamic architectures based on the used graph, representing the used components and links. The other useful analysis is the invariant analysis, which checks if a certain property holds for a number of configurations.

DynAlloy extends the Alloy language by allowing dynamic architectures to be represented as a sequence of states. The run clause is used to find the possible dynamic solutions and the assert clause helps find if a certain assertion is valid by trying to find counter-examples [92]. Dynamic Alloy is not the only extension to Alloy that attempts to support dynamic architectures. There are other developments such as Imperative Alloy [93], Electrum [94], and Dash [95] that can be used for this purpose. All these extensions are possible because Alloy's grammar is formal and exists as Antrl .g4 file and can be extended by users who want something more from the basic language.

6.4 Simulation of Models

We use modeling and simulation because systems are too complex to represent in any other way, so we can ensure that they work properly. Analytical representation is nice if it can be found to represent a system, but it is unlikely to be easy to find for a complex system. That is why simulation can help check models without having a complete analytical representation, for example, a set of differential equations. Another important part of the simulation is the representation of time, which is necessary for simulating dynamic systems, which are all physical systems interacting with the real world through sensors and actuators. Concerning time, systems can be discrete or continuous. The majority of CPS are in most cases continuous as they measure and control physical processes that are continuous. On the other hand, many multi-agent systems and their interactions can be modeled as event-based systems, depending on the type of model used. On the other hand, simulation can be deterministic or probabilistic depending on how the inputs and the simulated environments are perceived [96].

Since every model is an abstraction that presents certain aspects of the system, it needs to be tested against variations of its inputs and to have its outputs evaluated. The simulation provides an environment where tests can be run over ranges of inputs where the goal is to test the model with different data and to estimate its fitness [97]. This makes simulation a vital part of creating models as the complexity of models is very high, and their validity needs to be tested. Therefore each modeling language that can be used in practice has to have access to a simulation environment. Good simulators have facilities to generate input data streams, analyze output data, and provide visualization and logging capabilities for analysis. They also can allow for setting up multiple simulations using Monte Carlo techniques.

Simulation of multi-agent systems is a way to represent complex interactions between systems and thus predict the behavior of the entire system of systems. Since the discrete events simulation techniques are well-studied and easier to implement in such a complex scenario, they are often preferred in this case. Continuous simulation is also used in tools such as Simulink, which targets physical systems. Combined methods are also possible as systems may have discrete and continuous variables that interact [96]. All this depends

on the simulation language's capabilities and environment. In some cases, more than one simulation tool may be utilized.

6.4.1 AADL Simulation

AADL and other system modeling languages are used to mostly present static architectures. Nevertheless, the simulation tools need to be able to show all properties of the model. This is not always the case as simulation environments focus on certain aspects, for example, timing analysis or failure analysis . The simulation needs to provide tools that exercise all components of the model. Since the AADL model is driven by discrete events, the type of simulation is discrete. This is a bit of a drawback when the environment must be represented where continuous processes may need to be represented [98].

The idea of the simulation is to represent the system's behavior in every respect, including the operation of different modes. Since modes are the way dynamic behavior is represented in AADL, this is relevant to the topic of this study. The support of all annexes in simulation is another challenge, as the developers of annexes may not provide a specialized simulation tool. Lastly, handling time in the discrete event simulation approach is always a challenge with simulators. Issues such as synchronization between simulation time and real-time and how time progresses, in general, can hinder providing realistic simulation of the model [98].

The idea of a successful simulation tool is to provide better insight into all aspects of the model. An important aspect of the simulation is considering how to log and present the results. Since every component has important state variables that represent it at a certain time, this can be used to represent the state of the entire system at a certain point in the simulation. The information can be logged in a structured format that tools can easily parse. This approach can allow separate visualization and processing to be used. Unfortunately, the simulation tool must be tied to the modeling language and its capabilities. That is why the decision of selecting a modeling language is so important not just for the semantics supported but for the infrastructure of tools and utilities.

The AADL is a standardized language and has support from several well-established tools. Some of them are the Osate development environment and the AADL inspector .

Such development tools provide simulation environments integrated with them. This allows for quick verification of the model, typically before the implementation but also during any time when simulation of the model is needed. The model in AADL allows for the definition of processes. Each process can contain threads that can have different means of communication. The language also allows for the definition of hardware resources as processors and buses. There is support for different types of schedulers to do timing analysis of multi-threading applications. The AADL Inspector tool allows for the addition of different plugins. The Cheddar and Marzhin are the two most useful plugins for time analysis [99].

The Marzhin simulator is a multi-agent simulator. It allows for the random execution of threads from each agent in the simulation. Each agent can have a processor, multiple processes, and multiple threads in each process. The different processes are viewed as a partitioning scheme in time and space. The simulation environment has graphical visualization of the results and is presented in table format. The real-time analysis uses some parts of the Behavioral Annex from the AADL standard.

One benefit of the AADL language is the fact it is extendable through defining new annexes. Since the language focuses on the communication between processes and threads and the timing and failure analyses it does not support continuous dynamic simulation for physical systems. The AADL model is discrete, and it focuses on presenting the architecture. For this, it needs to be paired with a different modeling language like Simulink . An alternative is to extend the language with an annex as is done in this work [100] where AADL+ is proposed.

AADL+ is realized through an annex to the AADL specification. The main contribution is the ability to support continuous processes through the definition of differential equations. This introduces the equation clause in the language. Equations can then be embedded in components such as sensors and actuators as part of the model. The new analyses can be run in the Osate tools as it supports simulation environment textual and graphical representation of the model. The dynamic analyses can be run in Modelica as an intermediary file is generated to interface with Modelica . This type of model-to-model transformation is performed through mapping rules used to connect the AADL model to the Modelica model.

The timing analysis and other analysis supported by Osate can still be executed in the environment using the embedded analysis tools of Osate.

6.4.2 SysML Simulation

SysML was specifically designed to work at the system level, covering large systems with complex interactions. It extended some of the capabilities of UML and therefore it is more appropriate for embedded real-time systems. The existence of a mature specification of the language leans toward the creation of tools that can help the modeler. Such tools are commercial or open-source, although many do not have rich simulation capabilities. This may change as SysML Version 2 becomes mainstream and tools start supporting the new rich capabilities. The simulation of these capabilities is not standardized as different vendors and approaches are used. There are some efforts to use a standard approach for converting the SysML model to a simulated one by the query/view/transformation (QVT) standard [101]. The SysML language is not designed to support simulation. Therefore, model-to-model conversion to standard simulation standards such as the Meta Object Facility (MOF) is a possible direction to bridge this gap.

As SysML can be used for modeling in different disciplines, the semantics can be defined by the user in their specific discipline, for example, control or mechanical engineering. The challenge in such transformations is to precisely define the semantics for each type of formalism for each model and their interactions in the system model. This can be a problem in trying to simulate a system model that tries to integrate all models in the system. An approach to achieve meaningful simulation is considered in the literature by using a model to model transformation to SystemC and performing the simulation there [102]. Furthermore, code generation is also easier to do from SystemC as a model-to-code transformation. The approach uses a transformation language for SysML to SystemC model to model conversion through Atlas. The model-to-text transformation is done by ACCELEO which generates the SystemC to C++ code.

SystemC is just one possible modeling choice for simulation, but others solve the issue similarly. Another successful direction that has been pursued is transforming SysML

into a modeling language with simulation support, such as Modelica. To this end, the SYSML4Modelica specification has been created by the OMG . Another direction is to integrate SysML with Simulink and other tools that enable the simulation of algorithms. This though is not a way to enable the system capabilities of SysML, but to complement them with continuous simulation. The creation of MOF compliant simulation model allows for the DEVS formalism to be used as well as the tools based on it [101]. The DevSys framework standardizes the process through the SysML4DEVS specification. The framework is mostly based on Java and XML but does not restrict the user to add other languages. The transformation can target different domains, where DSL can be developed based on SysML. The latest version of SysML allows for easier creation of DSLs through extensions based on KerML [103]

6.5 Practical Approaches for System Generation and Maintenance through Modeling

The design of complex systems often involves multiple models addressing different aspects of the system. In some cases, they are developed in different modeling tools and different modeling languages. Model-to-model transformation can become important as we may want to consolidate models and do a simulation in common modeling language [104]. A transformation from SysML to Modelica or some other modeling language has become the main interest for many researchers. Such transformation use directed graphs to represent the different models. As different models have different semantics, a bidirectional transformation may not always be easy to achieve, although many works have explored the value of bidirectional transformations.

The representation of a model, independently of the modeling language, can happen by using standard representation such as XML Interchange language (XMI) . This is a structured approach to represent a model universally and to facilitate the use of tools to do transformation. The challenges when a transformation is performed are that there is a loss of information and different semantics in one modeling language versus another modeling or programming language. The significant problem is that the semantics of modeling languages like SysML and UML is not unambiguous as it is for other modeling languages or

programming languages and this is a problem for both model-to-model transformations and model-to-programming language transformations.

Sometimes a modeling language may not be particularly suited to the project. In this situation, the language may need to be extended to support new abstractions. An example of a language that allows such extensions is SysML which provides the so-called stereotypes. This allows for the creation of Domain Specific Languages (DSL) through profile-based extensions of SysML [105]. The described mechanism compensates for the generic nature of the SysML modeling language.

In the usual case, the modeling approach involves a unidirectional model to code transformation. This could involve code generation tools or other methods that create the implementation starting from the model. This is a top-down approach which is good at the initial phases of the design and development of the system. In later stages, though, very often, the implementation starts to deviate from the model. The reasons for that are many, but some start from the maturity of tools, the used process or the knowledge of different groups that do modeling or implementation. The capabilities of the implementation language are also much more diverse than the modeling language. It is interesting to bring the implementation in sync with the model, especially when dynamic behavior is expected. For this, a reverse process of code-to-model generation can be useful if supported.

The difficulty in reverse engineering production code to model languages is the semantic differences between modeling languages and implementation languages such as C++ . Such reverse engineering for recovery of the model can be done by creating mappings between the two languages [106]. The approach can be even more successful if the modeling language has well-defined grammar and semantics. Modeling languages such as AADL and SysML version 2 with their textual representation and formal Xtext grammar can be less ambiguous and can be more successfully used in this direction. Since both languages can be represented with Abstract Syntax Tree (AST) , the mapping can happen based on the associations of the elements from the different languages. The main challenge is handling cases where semantics from the implementation language cannot be represented with the modeling language.

In the hardware field, text-based languages have long replaced visual ones. Examples are Verilog , SystemC , and VHDL as they are the ones predominantly used for hardware design.

A method to reverse engineer VHDL to SysML has been proposed in very few works[49]. The approach has focused on generating SysML block diagrams from VHDL code, but it can also be used for any other SysML abstractions. Since VHDL is simpler than a standard programming language, this transformation appears practical. The transformation steps are analysis, parsing, mapping and model generation.

The reverse direction of code to model needs to be integrated into the development cycle. Figure 6.1 shows such an integrated process. The original model generates code for the system, which inevitably needs to be elaborated through manual coding as the system is implemented. A model is generated based on the real code using the reverse engineering approaches described above. Such automation can be helped by experts that help validate the code by using tools. As a result, the model is updated and the system now has this model as representing the true state of the implementation as much as it is possible, given the gap between the semantics of the implementation language and the modeling language.

The process shown in figure 6.1 allows for quick re-certification of the design through continuous modeling. The main premise is that the link between implementation and model is never broken. Ideally, the implementation language could also be a modeling language, but this does not sound realistic with the current level of abstraction in programming languages. The burden of making the connection between the two has to be on good tools for code generation, model generation, and model validation . The issue is that too generic tools may not apply to every domain without domain-specific extensions or customization.

In certain situations, this approach may be easier to follow. Such is the case when modeling languages with formal grammar are selected, such as SysML version 2 or AADL . On the other hand, code generated for frameworks with a fairly structured architecture can help too. An example is ROS2 , where the definition of the communication mechanisms is formalized through configuration files, and the code of nodes follows well-defined idioms and patterns. Therefore a bidirectional bridge between SysML and ROS2 is a promising direction for tool development.

A further refinement of the process of modeling for dynamic systems is shown in figure 6.2. The process starts with generating a system model following a top-down approach . After the system model is simulated and considered solid, the next step is to generate

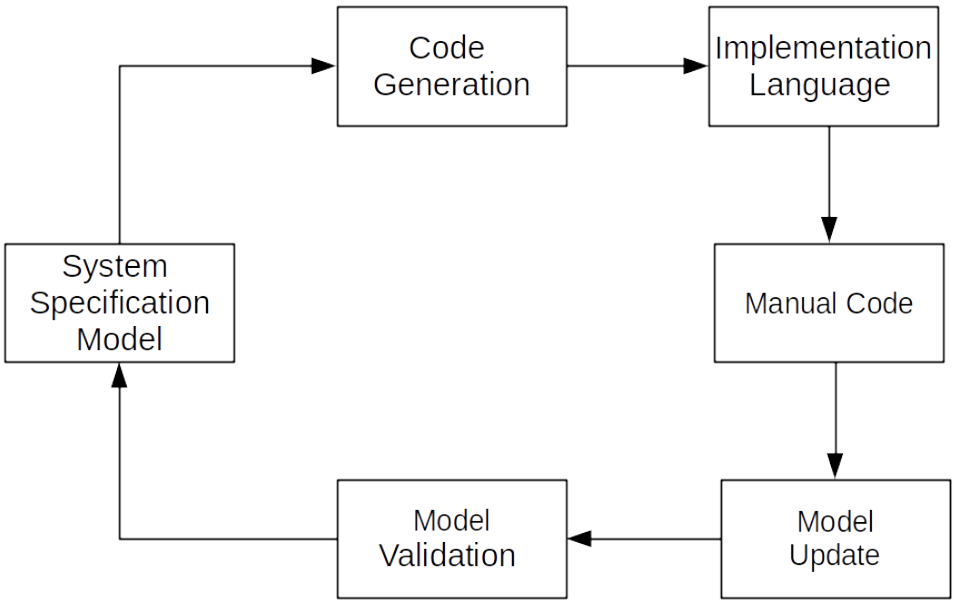

Figure 6.1. Model to Code Bidirectional Process

code for all components. The difference here is that in addition to generating code for each component, there is a model description of the component, which includes its interface and behavior. The model is attached to the source code in text format. For this, we can assume a text-based modeling language such as SysML version 2 or AADL .

Inevitably there will be code modification as the system needs to be developed or even after it is developed when maintenance and upgrades are necessary. The component model will be updated to match the code changes if they are relevant to the model and break any promises. The updated model for each component will be automatically validated. The following step will take each component's model, assemble the system model again, validate it, and ensure it still works. If it works then the system model will be updated with the changes in the model.

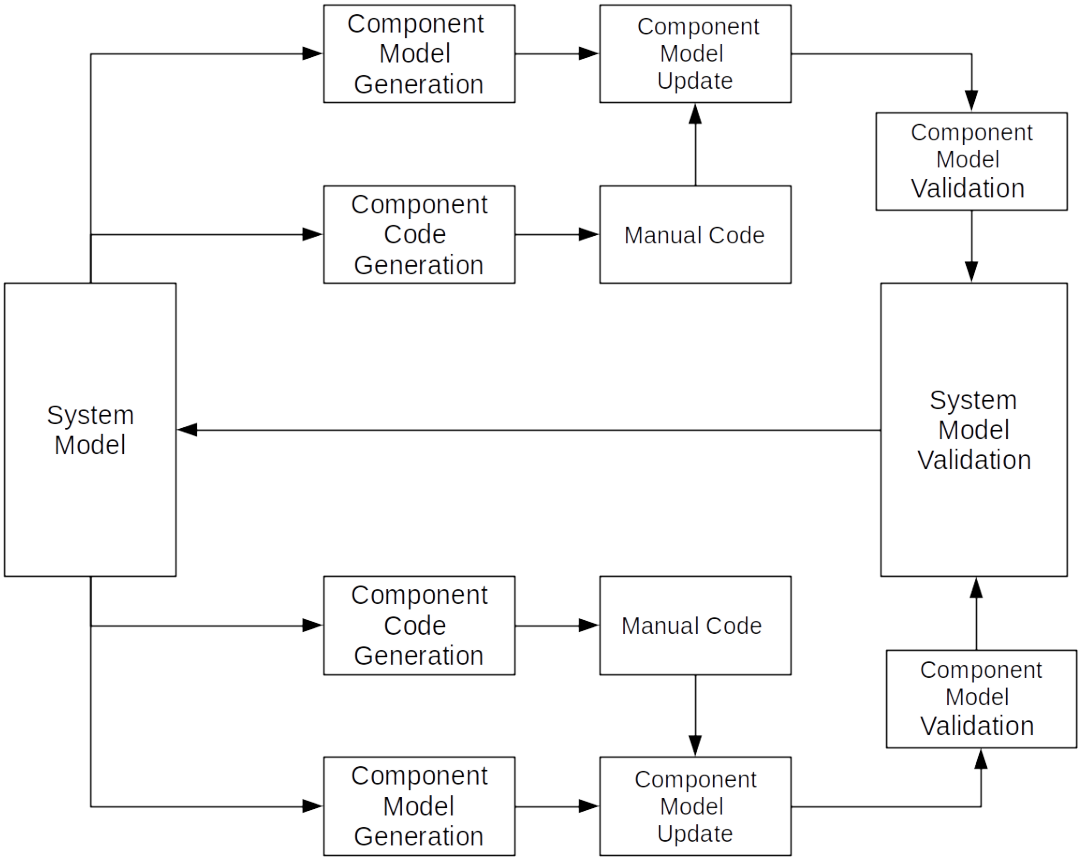

Figure 6.2. Component Centric Model-Code Integration

On figure 6.2, the design-time is shown, but a similar model validation scheme can be utilized during run-time if a component's code is modified as well as its model, then a system model validation can be performed before the component is allowed to be used in the system. The system model validation can serve as run-time assurance at a high level with certain guarantees as much as they can be achieved through the expressiveness and granularity of the model. The approach is not a one hundred percent replacement for testing but is a step towards dynamic validation of newly added parts of the system. Techniques such as in-place testing can complement the approach to provide even further guarantees before the dynamic change gets the green light.

The scheme discussed so far can be successful only by adopting appropriate specialized tools. The generation of model and code for each component and their arrangement in the

code needs to be human and machine-readable. The update of the model, even if done manually, needs to follow a particular process so that the updated model is still machine-readable and has to have the correct syntax. The assembling of the individual component models will also need to happen automatically. The same is true for the model validation.

6.6 Conclusion

MBE has made significant progress over the last several decades. In many fields of the industry, it still has not progressed enough due to some of the challenges mentioned in this chapter. The use of modeling when designing systems with dynamic behavior is promising, albeit challenging. The trend towards increasing the role of modeling in software engineering is accelerating, and this applies to all phases of product development, from requirements to maintenance and diagnostics. One of the critical issues is the connection between models and implementation code and their interaction through all phases of product development. Such connection is central to the success of modeling, used for dynamic systems throughout the entire development cycle.

7. DYNAMIC RECONFIGURATION OF SYSTEM OF SYSTEMS

7.1 Specifics of System of Systems with Respect to Dynamic Reconfiguration

System of Systems (SoS) are increasingly being used in many industries, although their origins come from military applications [107]. Other terms that are related to Sos are Large Scale Systems (LSS) and Ultra Large Scale Systems (ULLS) . The commonality between all these classifications of systems is that they usually require reconfiguration as systems join and leave or get updated while the SoS operates. There are many scenarios, from search and rescue operations to IoT applications, where ad-hoc configurations can be formed based on changed environmental conditions or the participating systems [108]. In such scenarios, time is always of the essence, and the systems can be safety-critical . These factors require a secure way of reconfiguration that guarantees safety, security, and functional integrity. Dynamic reconfiguration is, an implicit expectation, although it needs to be treated as an explicit requirement.

SoS have numerous characteristics that make them very difficult to reconfigure, although they expect a dynamic environment more than standard IT systems. These characteristics are well-described in the literature. They include operational and managerial autonomy, emergent behavior , geographic distribution, and evolution of the SoS [109]. These very characteristics encourage dynamic changes, although the lack of guarantees that come with it is a major research challenge. The main hurdles come from the lack of standards and the different organizations that participate in developing the constituent systems. The development of each system in separation is not the actual problem but the process does not include standardization in how to communicate with other systems and accommodate dynamism.

As in many other engineering disciplines, one approach to tackle a complex task is to handle complexity by simplifying the interactions and making the interfaces more specific and detailed. If systems are developed in isolation and adhere to different standards, then it would be hard to integrate them with certain guarantees of quality, and results will always be unpredictable. Therefore one promising direction is to set some standards for interfaces between systems, independently of their origin and implementation details. Such an approach

can be made through common representation as in examples of model-based engineering [110]. Another element of compatible designs is to use interfaces that offer universal ways to make the systems available to other systems. Systems can provide services as a common paradigm through their interfaces to build compatible distributed systems.

The individual systems that compose a system of systems are called constituent systems (CS) . A vital behavior of constituent systems worth mentioning is that each one has its state and perception of the world independently of the other constituent systems. This makes the possibility of a global state a non-feasible reality. For generality, we can assume that there is no central system that has a full view of all CS, but rather every system is autonomously making decisions that best achieve its own goals. Even if there are classes of systems with central control, they are still rare, and their evolution is not so easy as it depends on the ability of the central system to grow with the growth of the SoS [107]. The analysis presented in this work focuses on distributed architectures without needing a central governing system, as this is a more general and flexible approach. This class of SoS is known as collaboartive.

7.2 Design and Deployment of SoS

The design of SoS does not follow a universal methodology as the field is still developing, and a great diversity of systems can participate in an SoS. This provides a lot of freedom and opportunity for innovation in this growing field. As the complexity of such SoS is high, a great many papers discuss Model Development Engineering (MDE) as a promising trend for coping with the existing challenges [54]. MDE or Model-Based Engineering(MBE) has the benefit of abstracting complex system concepts, and this alleviates the burden of growing complexity and helps the designer focus on different aspects at a time. MBE is becoming more popular, and the growth of SoS is accelerating this trend which brings opportunities for a symbiosis of these domains.

The numerous combinations of system interactions limit the feasibility of traditional design-time approaches when working on SoS. Nevertheless, static design architectures have been studied extensively and widely accepted when regular systems are used. Therefore we will start by exploring static approaches, along with their benefits and limitations. It is worth

noting that each system may also be a SoS in itself and have some dynamic properties, so the purely static approach may not always be feasible unless applied in the right context [109]. Systems can have dynamic behavior through the components used to create the system. This brings the dynamic aspect in and out of the system itself.

The deployment of systems can be done very infrequently as done traditionally or on a more frequent basis as in the case of quickly formed system of systems in DoD or search and rescue scenarios. We can consider changes to CS to happen at run-time or during a system redesign. The run-time approach assumes special mechanisms that support run-time adaptation through modifications of certain parts of the system. These operations bring up issues such as safety, security, and stability of the deployed SoS . In addition, the reassurance of such systems is a challenging field that is a developing research area. One of the main challenges is the complexity and expression of the dependencies that arise from dynamic changes.

7.2.1 Design-time Approaches

One of the characteristics of a SoS is the dynamic behavior at run-time. The complex interactions between many systems can easily lead to an explosion in the complexity and the state space and can become inconceivably hard to tackle. Some authors suggest restricting how systems can interact with each other [111]. These interactions can be represented through Interface Block Diagrams (IBD) in SysML to model the possible configurations between the constituent systems. This way, a set of possible configurations can be used at run-time.

To solve a complex problem, one can try to restrict all possible choices through the reduction of freedom that the design has. This approach focuses on the explosion of possible combinations of connections, interfaces, and services. If each architecture supports a known and limited number of configurations, then it is feasible to create code to support each one during the design phase. If modeling is used, this can happen through code generation from a validated model reflecting each configuration. Each system, as part of the SoS can undergo the same approach. This can be time-consuming but generates validated systems that sup-

port a finite number of configurations. These systems can then be used in combination to create a dynamic SoS. If this type of restricted dynamic behavior fits the requirements of a SoS, then this solution is a sound and practical one. The process is illustrated in figure 7.1.

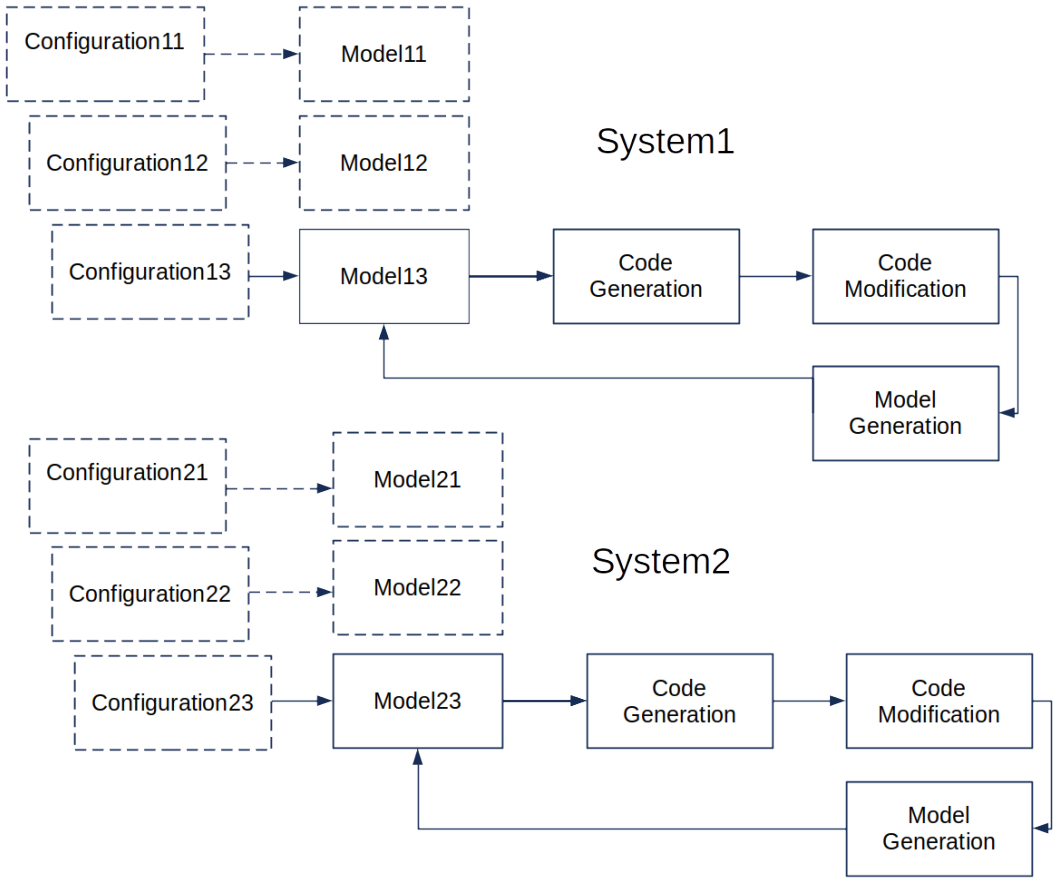

Figure 7.1. System Design Process [112]

Different combinations can be selected during the design to check whether they work well together. As a result of this process, only valid combinations can be marked as successful. For example, Model11 from system1 can be checked against model21 from system2 and model 31 from system3. Only these combinations can be allowed to run in the SoS . The method has a disadvantage when there are many combinations, as it can quickly become impractical. For some system of systems, though, it can be a very reliable and secure method as it is done at design-time with more rigor.

The concept of designing a model that encompasses the existing models is called meta-modeling [109]. In the SoS context, meta-models that take the models of the constituent

systems and validate the meta-model have much potential. The most important challenge is to reflect the systems' changes into their models to represent the evolution and dynamic changes in each constituent system. Another challenge is that each model may represent the same things differently, and consolidation of the models may be required before attempting meta-modeling. This can help with dynamic assurance, which is ultimately the goal of having some control over SoS evolution.

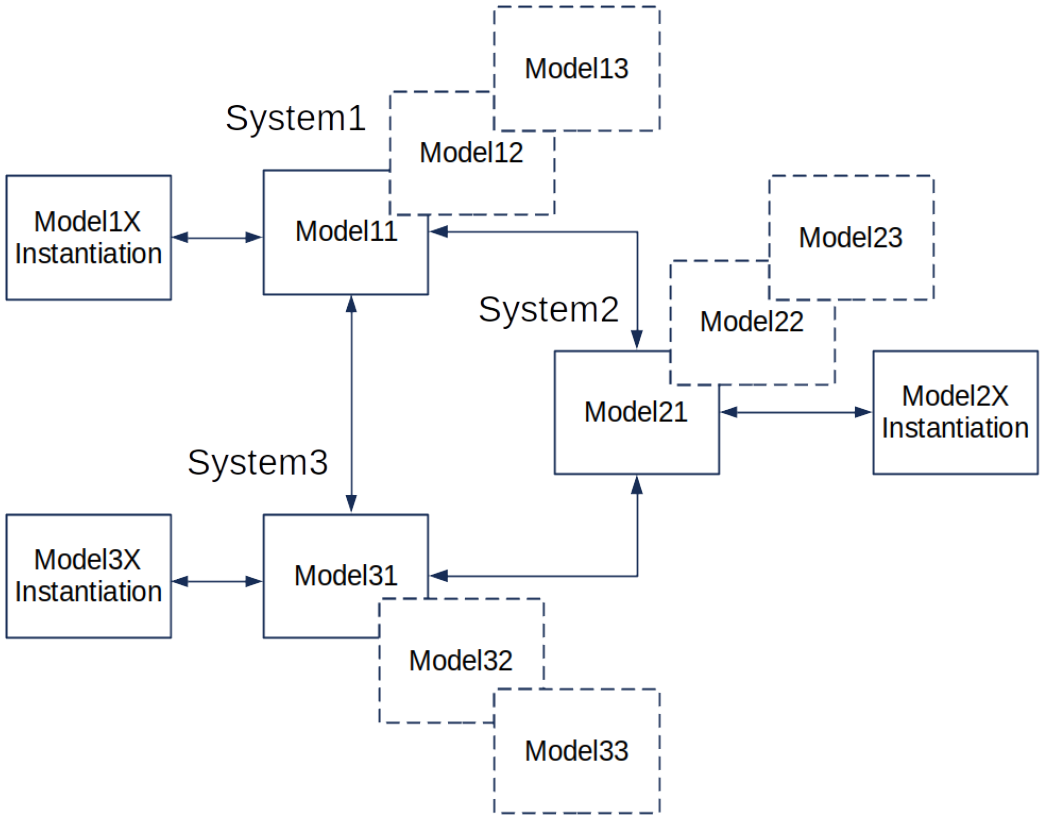

Figure 7.2. SoS Design Process [112]

Generation of SoS

Model-based engineering can be very useful for validating SoS architectures as well as in other phases of their development. The model by itself is not going to reach its full potential if it is not connected to the real implementation of the system. This makes automatic code generation from models a promising direction for SoS. The connection of code to model

is a challenge that is especially difficult in going from code back to the model. Therefore the code-to-model relationship is an essential concern in the evolution of SoS . The use of specialized tools can be very instrumental in this direction so that it can be useful in practice. The main challenge is the diversity of modeling languages and implementation languages and the representation of semantic differences.

Model-based development aims to help cope with complexity and raise the level of abstraction. This appears very appealing in the large-scale system arena. When using code generation from code, one can also avoid some coding inconsistencies that traditional techniques can't guarantee. Another reality that needs to be faced is the addition of manual code to complete the implementation of the system and how this maps to the model. One such modeling language is VDM-RT , targeting distributed embedded systems with abstractions for an object, cpus, and buses [113]. This language leans towards validation and simulation of the modeled system before the code is generated.

In some cases, a model-to-model generation is possible when one modeling language is at a higher level and serves different objectives than another one. Such works have tried model transformation, for example, from SysML to AltaRica [114]. The idea behind this is to get to a model language closer to the implementation and to model additional aspects of the system. With newer versions of SysML, specifically version 2, the textual representation allows for creating many different concepts, such as state machines and others. This makes the generation of final programming language code from the SysML model more manageable and more straightforward compared to going through another modeling language, although the chasm between a high-level modeling language such as SysML and the different programming language implementations for different systems is a challenge that requires a lot of new research ideas.

For example, automatic code generation has been proposed from different modeling languages such as UML . The generation of code from state diagrams has been studied extensively [48]. The approach represents the systems as objects, and each object can have a Finite State Machine (FSM) . Each FSM is controlled through events that come from other objects or externally. The difficulty in using UML comes from the fact that UML represents the state diagrams in a graphical form, and this makes it harder with the implementation of

automatic tools. Code generation is still focused on a particular small part of a system or even a component, and tools and techniques do not cover the generation of code for SoS.

A universal method for deploying SoS is based on services as a universal mechanism for building distributed systems. A system of systems can be considered a collection of devices that present services, where each service presents a service interface. Such an approach [115] allows for the building of SoS dynamically by connecting devices through services offered by other devices. The proposed method uses an SoS composer based on a graph model of the SoS, which helps match devices with the services they need. The IoT domain is ideal for this approach as IoTs are added and removed, and new services can also become present or disappear. This approach does not focus so much on the implementation but on the capabilities and interfaces of the CS as part of the SoS.

7.2.2 Run-time Approaches

In order for run-time changes to be possible, each system has to be equipped with the capability to connect to other systems and to present its services as they change dynamically. As presented in figure 7.3 it can be practical to dedicate a control component as part of each system that takes care of this job. This way, the remaining components of each system can do their job to provide the local mission of the constituent system, and the control component can be in charge of reorganizing them and presenting the updated system interface to the other systems. Control components can comply with a common interface standard so that independently developed systems can be easily integrated into different configurations. Standardization in this area would be very beneficial in the long run, although it may be difficult to proliferate because of all sorts of legacy systems.

Each constituent system can also have a state machine that reflects its overall state, which is a function of the participating subsystems and components. Based on the state, operations can be allowed in a safe manner or not allowed so that the system can work as intended. The global SoS does not have to present a global state explicitly, as it depends on its constituent systems and their state. The global state shall be avoided as we seek the

independence of individual systems and their independent growth. If some systems do not have a state, then the integration is even easier, although this would rarely be the case.

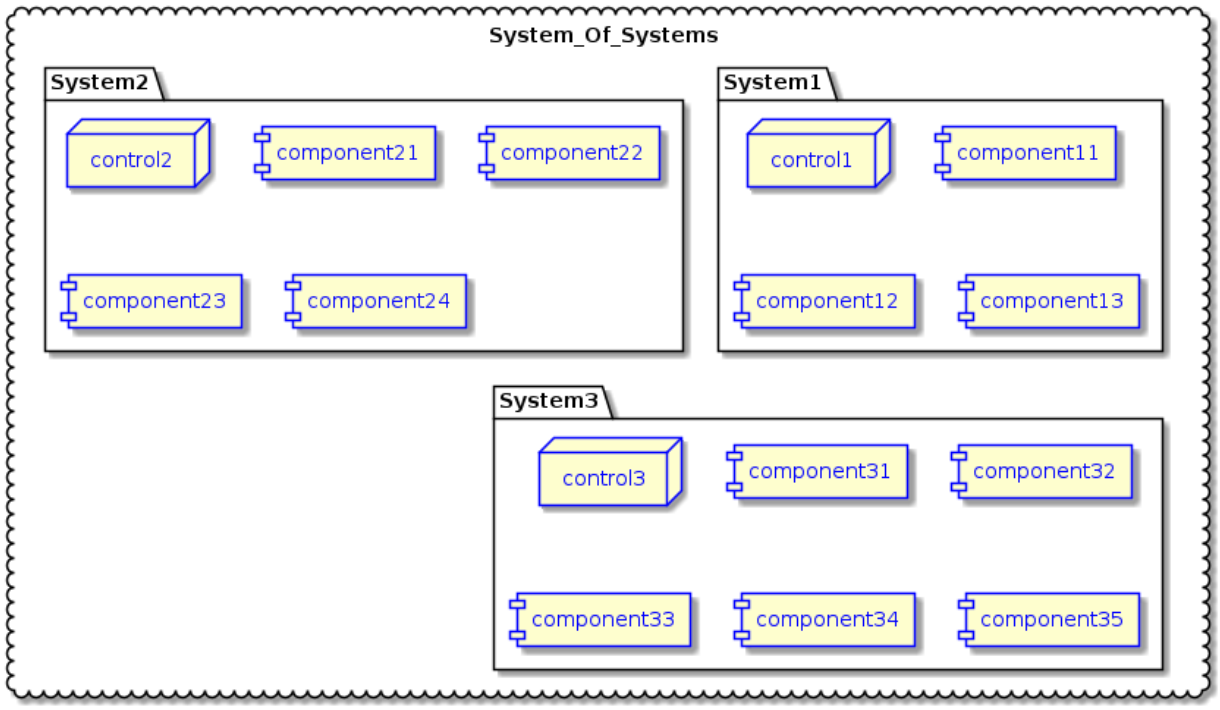

Figure 7.3. SoS Control Element [112]

The ability to control individual component systems so that they can be added, removed, or updated to the SoS is essential. We can think of each constituent system as equivalent to a component in a regular system. In that respect, each system should have an interface representing it to other systems. The more well-thought-out and standardized this interface is, the easier it will be to assemble SoS. Finally, the ability for systems to propagate and advertise their capabilities to other systems is another requirement for each CS that needs to support run-time dynamic behavior.

Rapid Development of Systems

In some domains, rapid deployment of systems is a need that is fairly demanding but necessary. Rescue and search missions, catastrophe response, and military operations all have such requirements of assembling a system on-the-fly as the situation demands [108]. Even in areas such as Industry 4.0 and IoT, there are requirements for rapid reconfigurations of systems. In such situations, time and quick guarantees are paramount, as there is no time to go through the traditional development cycle. The only approach is to configure a system at run-time based on the knowledge of the constituent systems and the run-time analysis that can be done quickly. This type of situation puts run-time assurance as a central piece in the puzzle of the validation of system of systems as they are formed.

The crisis response field has numerous examples of systems of systems that need to be reorganized dynamically. They are also referred to as Smart Crisis Response System of Systems (SCRSoS) [116] in the literature. Their dynamic behavior assumes dynamic links between systems being formed and services being offered, depending on the environmental characteristics. Switching to new behavior can be precipitated by events triggering changes from the environment and the state of some of the constituent systems. The main focus in achieving such quick reconfiguration capabilities of a SoS can happen through very good cooperation of its constituent systems.

One approach for reconfiguring a SoS based on events is associating roles to each constituent system. There are formal approaches to specify such reconfiguration behavior. One such approach uses the Maude language [116]. The Maude language has support for dynamic changes applicable to SoS in general, particularly for SCRSoS. The ArchSoS extension of Maude is an Architecture Definition Language appropriate for representing SoS and their dynamic evolution.

Assurance of SoS

The assurance of SoS is a complex subject, and there is no universal solution that can work everywhere. It can be done offline or during run-time , although in both cases there are significant difficulties in how a SoS is assured. Many validation and verification (V&V)

techniques that use tools such as ACSL are oriented to verify the behavior of individual functions [36]. In some extensions of ACSL , like MetACSL , a set of meta-properties are validated, and then a set of functions are included [117]. This is still a very limited approach for SoS where the scope is much larger, and the implementation between different systems can vary substantially. This means that a new approach is needed to achieve a higher level of assurance for systems at scale. The same concept could be extended to a set of components and systems instead of a set of functions, as in MetACSL.

The offline assurance methods rely on techniques that explore the state space and can take a really long time and can become too difficult when applied to complex systems. Run-time verification on the other hand monitors state transitions as the system executes [118]. If a discrepancy is found from a specified spec, the system can react in a specified way. This approach is fine when the system is well-defined and does not have dynamic reconfiguration. When the number of new reconfiguration sets are known, a set of respective system specs can be used for each one as an interim solution. A complete solution for an arbitrary dynamic reconfiguration requires a new approach since the combinations are many and unpredictable at design-time .

As suggested in [118], a Domain Specific Language (DSL) such as Genom is a convenient way to present the specification of a system. Such language can have formal grammar and, therefore, can take advantage of automatic tools to parse the grammar. The DSL can specify the system, and then the final system can be generated through code generation. A special case is when the DSL allows modeling, and different properties of the design can be checked at any point. Components and entire systems can then be specified and generated after their properties are checked. If the models are kept up-to-date with the system changes, then assurance is possible at run-time . This approach is explored in other chapters of this work. Realistically, offline and run-time verification are needed for the assurance of complex SoS.

The idea of expressing the system with something that is similar or the same as the implementation language is explored in the following work [119]. The authors argue that the specification, the implementation, and the assurance should be done with the same language and tools. Within the SoS domain, where we may have different implementation languages,

we can only guarantee that the specification and verification are done in the same language. For this, a plausible approach is to use the same language for specification and verification and modeling and to generate different implementations based on the target language. As long as there is a bidirectional correspondence between the implementation language and the assurance and verification language, then we can argue that run-time verification is possible for SoS .

This scheme is shown in figure 7.4, where different implementation languages are assumed for the three constituent systems. In order to be able to do assurance, we need to represent each system in a common modeling/specification language. For this, a model-to-implementation bridge is needed for each implementation language. The approach is flexible as new implementation languages can be added, and only a new bridge will need to be developed from the new implementation language to the model. Therefore scalability and generality are achieved by keeping a common modeling and specification language for each system, for example, SysML . This allows for meta-modeling for the entire SoS as shown in the diagram.

The challenge of the shown approach is the bottom-up direction of updating the model from the implementation languages. Since every implementation language has different semantics that needs to be mapped to the model language, different mappings need to exist for each implementation language. This process also will need to lose some semantic features in the process. The final challenge is to ensure that the models of the individual systems need to have the same formalisms so that the meta-model can be formed. It is also possible to envision that the models of the individual systems are using a lower level of modeling language that needs to be mapped to a higher system-level meta-model.

One direction of providing assurance is to shift parts of the assurance to run-time , not only design-time as the more traditional approaches prescribe. This is required for highly dynamic systems because of the large state space of possible configurations that cannot be assured at design-time. For this, new paradigms, such as the system's run-time models, need to be used. The B-Space framework is such an attempt [120]. The framework's focus is to provide safety assurance at run-time through comprehensive models. There can be a plethora of models addressing different aspects of the system, as one model cannot be comprehensive

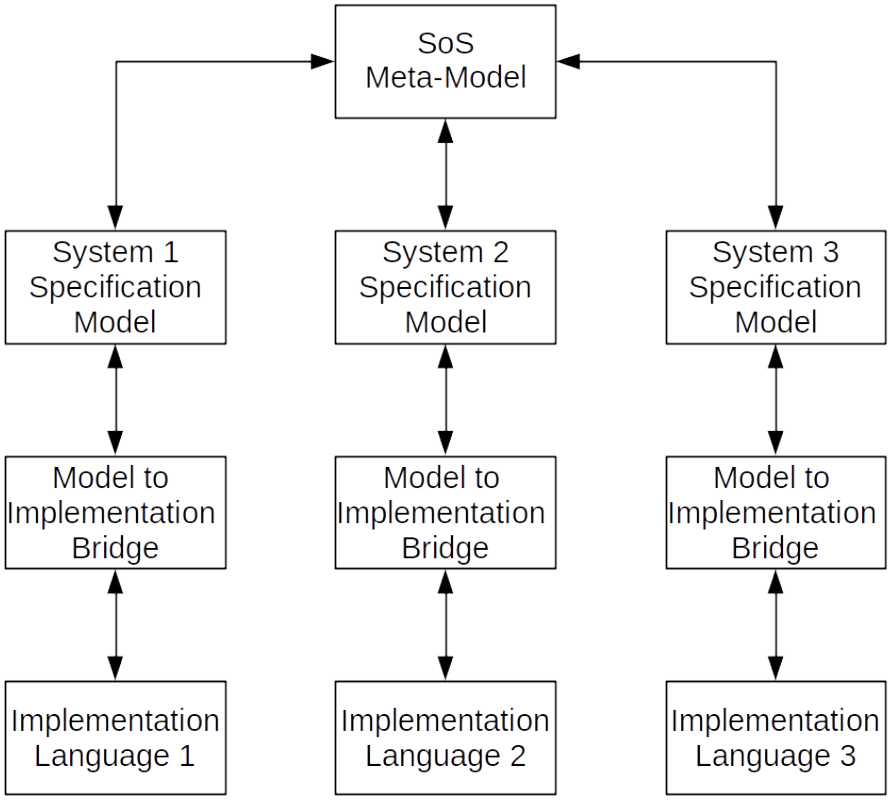

Figure 7.4. Model to Implementation Diagram

enough for all design aspects. The collection of these models is referred to as B-Space, as the models represent the system in virtual space.

Dynamic architectures can be formed by systems of systems requiring different services and forming new dependencies after instantiation. As these architectures are formed, conditional safety certificates are introduced to ensure safety conditions are not violated. These safety contracts are called ConCerts [120], and their function is to provide guarantees to the participating systems. The realization of ConCerts can happen through formal grammar, as this allows their integration into existing modeling platforms and tools. They can be viewed as additional entities in the architecture that can help with the run-time reconfiguration of SoS while preserving certain safety properties. The approach tackles the run-time state space enormity by introducing safety guards focused only on certain aspects and did not need to

know the entire state of the SoS. In other words, they act as intelligent blocks that are a second architecture alongside the main system architecture. Since they are distributed and agnostic to any particular place of residence, they can be applied to different configurations, hierarchies, and topologies.

As in many other engineering disciplines, one way to tackle a problem is to use the principle of separation of concerns. For example, the functional requirements differ from the non-functional ones, and safety is in its own plane. This approach can be used in the design of systems where safety and its run-time assurance can live in a parallel architecture that focuses on just one thing. Keeping the run-time assurance architecture separate has a lot of benefits, one of which is the ability to re-certify it easily if changes are needed and to use different design techniques and tools [121]. A side benefit is that system's complexity is on the rise, and certifying the main system may not be feasible, especially with the use of AI and other computer-intensive algorithms in contemporary systems, for example, in robotics. The separation of concerns principle allows for keeping systems safe, even if they contain some unsafe parts. The benefits are even greater in systems with dynamic behavior, such as SoS, where analyzing all possible scenarios may be impossible. Simpler run-time assurance becomes an alternative path to the traditional design with high returns on the invested time and effort.

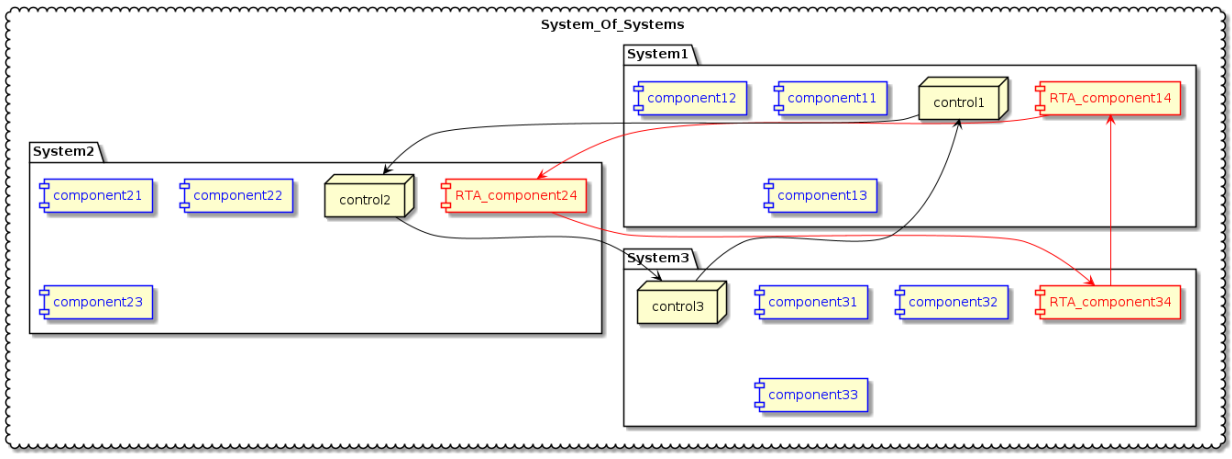

Figure 7.5. Run-Time Assurance Architecture Embedded in a SoS

An example of run-time assurance architecture embedded into a system of systems is shown in figure 7.5. Each system has a Run-Time Assurance (RTA) component dedicated to run-time control. The RTA components are connected through an interface specific to the overall RTA of the entire SoS. This can guarantee that if one of the RTA components senses a violation this can be communicated to the other RTA components. The diagram shows a parallel connection between each system's control and diagnostic components. The shown architectural approach shows how a SoS can be presented and controlled in different layers: control, assurance, and function. This type of approach can be thought of as compositional run-time assurance as we include each RTA portion from each constituent system.

The RTA components can use contract-based assumptions and guarantees [121] to make them amenable to formal verification. Since RTA components determine the safety of the system, they need to be expressed in a formal language that can use some formal verification techniques. The interfaces between different systems' RTA components can also benefit from standardization. The same is true for the control components shown on the diagram. These different logical networks can give new life to SoS implementation and development. Even if the approach can be standardized for a particular domain, this is still a significant improvement in the way SoS is architected.

Mission-Based Validation

Many authors start the argument for system validation by validating the mission [116]. This is the ultimate high-level goal that needs to be achieved for a SoS, and mission-driven testing and validation is a popular approach. This is a fairly high-level approach and can abstract itself from the actual SoS composition, especially in the context of dynamic reconfiguration where the SoS can vary at run-time . Mission-based validation is still part of the overall validation effort. Even if it has a narrow scope, it is a form of cross-cutting concept that helps achieve a reasonable goal faster.

The SoS can be described by its mission and the missions of its constituent systems. Some works have proposed the use of special languages for specifying goals in order to represent the mission of the SOS. One such language is based on the KAOS requirements language

[122]. The global mission of the SoS describes the SoS, and the individual missions can be very different, but they still need to help to accomplish the global mission. Various mission description languages can be used for mission-based validation . Some of them are based on XML or similar meta-languages, and others are specifically developed for this purpose.

Methods to validate the mission characterize the mission as a sequence of actions the system takes to achieve the goals. The mission can be controlled by resuming an action, going back to a previously known successful action, or can be completely aborted. The authors of [89] advocate for the creation a mission language to represent and monitor the mission. Each action can be individually validated by checking its preconditions and attributes. Automatic code generation of monitors for each action can be performed based on the mission specification. This approach does not consider the actual design of each system but rather the execution of the mission and focuses on its validation . This can be considered a high-level, black-box approach for validating complex systems from the perspective of the final result - the mission execution.

Large-scale systems that need to be quickly assembled in order to achieve a specific goal are named mission-oriented system of systems. The challenge of such SoS is that they need to guarantee interoperability [123]. The interoperability analysis can start at the modeling phase, focus on the operational activities, and then consider the technical capabilities that can enable the mission. A better way to describe the mission is to represent it as a collection of activities and associated information exchanges. In addition, one needs to set criteria for the success of the mission, for example, the time for completion and successful achievement of the mission's goals, for example, destroying the target or reaching the destination and delivering a package. The operational activities need to be considered as requirements for the technical capabilities. One area that requires some more special attention is the communication between systems and their ability to perform actions based on communicated information. This analysis can include the probability of communication or execution failures that can lead to mission failure.

7.3 Conclusion

Dynamic reconfiguration of single systems is challenging as it is. It is even more challenging for SoS. The relatively new field is part of the problem, and another issue is the fact that SoS is constantly developing with shifting definitions and expectations. This also brings research opportunities and challenges, as described in this chapter. One of the most challenging directions is the run-time assurance (RTA) of SoS, as complexity and diversity defy existing design and analysis methodologies for standard systems. As described in previous chapters, growing momentum in the MBE approaches brings a lot of potential for the future of SoS. This chapter presented some potential directions for addressing SoS dynamic behavior in general.

8. DYNAMIC ARCHITECTURE DESCRIPTION LANGUAGE

8.1 Motivation

As the research of the literature in this study showed, the existing ADLs provide facilities primarily for static design. They are not very suitable for the type of systems that will be needed in the future. The modeling is mainly done at design-time, and the run-time behavior is usually fixed. The advent of IoTs and self-configuration sensor networks will exacerbate the need for run-time changes in the behavior of systems. The need to move modeling from the design to run-time is a trend that is new but already well-established. When we look for a solution in a new domain, there are usually two paths: to change and augment an existing solution or to create a completely new one. Each approach has its pros and cons. The first approach is certainly possible as many modeling languages, such as AADL and SysML , support extensions. On the other hand, a brand-new language can thoroughly approach dynamic behavior as a central design feature and provide a cleaner approach. This can also be used in conjunction with existing modeling solutions to add the missing functionality.

As many of the modeling languages have semi-formal nature, it is not easy to provide precise semantics without creating a new DSL extension, and even then, the remaining part of the language is still semi-formal. Another concern when using an existing language is the complexity of the language and the tools it comes with. A more straightforward and targeted solution is sometimes a better and more elegant way that can be easy to adopt and use. These reasons lead to choosing the approach of designing a new dynamic run-time ADL called dynADL . This new architecture language has been influenced by several existing modeling and interface languages that cover different aspects of design, graph notation, interface definition, and contracts. These languages are presented with features that are of interest to the creation of dynADL.

8.2 Influencing Architecture Description Languages

Developing a new Architecture Description Language (ADL) is a serious undertaking. Doing all the work from scratch is exciting but very time-consuming and unnecessary. For

this reason, we want to look at existing languages that do certain things well and explore the possibilities of reusing some of the ideas. Since our new ADL requires architectural representation and handling, we can look at the Graph Modeling Language (GML) language, which specializes in graph parsing and visualization. A central part of dynADL is the representation of interfaces for components and systems. The Thrift specification, being an Interface Definition Language (IDL), presents interfaces for messages and services in an elegant way and can be used as a base to add other interface elements. A language like ACME can serve as an example of how complex architecture is defined. Things like components, connections, and ports can be reviewed from the point of view of the new dynamic ADL. AADL is also considered because of the same reasons as ACME. These languages and others provide specializations in how they address specific aspects of the design, and some of their features can be used as influences to our development. They all do not address run-time behavior explicitly that we would like to support in dynADL.

8.2.1 GML

GML stands for Graph Modeling Language. It has a format that is simple, flexible, and expressive and has been used for some time. Its grammar exists in Antlr format, which allows for features to be reused easily. It uses a simple text-based syntax to represent graphs. The main keywords that are used are : *graph, node, edge* [124]. The language is based on key-value pairs where the key is a keyword from the language, and the value can be anything. GML provides even too much freedom in the choice of keys and values. The node is something that can be extendable, and this is very convenient as nodes can be used to represent software components or systems. A simple example is shown in listing 8.1 [124],p.1.

8.2.2 Thrift

Apache Thrift is an open-source framework that is built to support distributed applications. It presents messages and services in a very elegant way through an Interface Description Language (IDL) that is formally defined, and it has a grammar in Antlr. It supports several programming languages, and it is relatively lightweight compared to other IDLs. It

Listing 8.1. GML Example

```
graph [
   comment "This is a sample graph"
   directed 1
   IsPlanar 1
   node [
     id 1
     label "Node1"
   ]
   node [
     id 2
     label "Node2"
   ]
   node [
     id 3
     label "Node3"
   ]
   edge [
     source 1
     target 2
     label "Edge from node 1 to node 2"
   ]
   edge [
     source 2
     target 3
     label "Edge from node 2 to node 3"
   ]
   edge [
      source 3
      target 1
      label "Edge from node 3 to node 1"
   ]
]
```

also supports serialization for data and the concept of services. Service Oriented Architecture (SOA) systems are attractive in the cloud and other areas where (SOA) is needed. Thrift has pretty good productivity when deployed, although the main interest in this work is the definition of interfaces that consist mainly of messages and services. An example of the Thrift IDL, adapted from [125],p.430 is shown in listing 8.2.

8.2.3 AADL

AADL supports both graphical and text-based modeling, and its grammar is defined in Xtext format. The development environment of AADL is based on Eclipse through specialized plugins. Although it has been created as a design-time modeling language for predominantly static architectures, it has some support for dynamic behavior through modes. It also has an excellent representation of processes, threads, components, and mechanisms to communicate between them. Its Java-based development environment provides tools for modeling and simulation in different operating system environments. Some of the constructs of the language, like components and connections, can be important to the development of this work. A representative example of the language syntax adapted from [126], p.24 is shown in listing 8.3.

8.2.4 Acme

Acme was created decades ago, although it has not been as popular as AADL or SysML, primarily because of the lack of good tools. It is a well-designed ADL that can be serve as inspiration for this development. In Acme, systems are represented as graphs with components. Components, connectors, and ports are central to the language and many other features that are less connected to this work. Acme has a formal grammar defined in Antlr, which is really useful as our development is also using Antlr. The language's syntax is somewhat complex and covers many different aspects of designing systems. The ideas in ACME can be used by simplifying the syntax and using only the relevant concepts for dynamic behavior. An example from the Acme language [127],p.57 is shown in listing 8.4.

Listing 8.2. Thrift Example

```
enum Market {
        Unknown = 0
        Portland = 1
        Seattle = 2
        SanFrancisco = 3
        Vancouver = 4
        Anchorage = 5
}
typedef double USD
struct TimeStamp {
        1: i16 year
        2: i16 month
        3: i16 day
        4: i16 hour
        5: i16 minute
        6: i16 second
        7: optional i32 micros
}
union FishSizeUnit {
        1: i32 pounds
        2: i32 kilograms
        3: i16 standard_crates
        4: double metric_tons
}
struct Trade {
        1: string fish
        2: USD price
        3: FishSizeUnit amount
        4: TimeStamp date_time
        5: Market market=Market.Unknown
}
exception BadFish {
        1: string fish
        2: i16 error_code
}
exception BadFishes {
        1: map<string, i16> fish_errors
}
service TradeHistory {
        Trade GetLastSale(1: string fish)
        throws (1: BadFish bf)
        list<Trade> GetLastSaleList(1: set<string> fish
        2: bool fail_fast=false)
        throws (1: BadFish bf 2: BadFishes bfs)
}
```

Listing 8.3. AADL Example

```
process control_processing
features
input: in data port sensor_data;
output: out data port command_data;
end control_processing;

process implementation control_processing.speed_control
subcomponents
control_input: thread control_in.input_processing_01;
control_output: thread control_out.output_processing_01;
end control_processing.speed_control;

thread control_in
end control_in;

thread implementation control_in.input_processing_01
end control_in.input_processing_01;

thread control_out
end control_out;

thread implementation control_out.output_processing_01
end control_out.output_processing_01;

data sensor_data
end sensor_data;

data command_data
end command_data;
```

Listing 8.4. Acme Example

```
System simple_cs = {
        Component client = {
                Port sendRequest;
                Properties { requestRate : float = 17.0;
                        sourceCode : externalFile = "CODELIB/client.c"
                }
        }
        Component server = {
                Port receiveRequest;
                Properties { idempotent : boolean = true;
                        maxConcurrentClients : integer = 1;
                        multithreaded : boolean = false;
                        sourceCode : externalFile = "CODELIB/server.c"
                }
        }
        Connector rpc = {
                Role caller;
                Role callee;
                Properties { synchronous : boolean = true;
                        maxRoles : integer = 2;
                        protocol : WrightSpec = "..."
                }
        }
}
Attachments {
        client.send-request to rpc.caller ;
        server.receive-request to rpc.callee
}
```

8.3 Language Development Technologies and Tools

A great variety of tools can help develop an ADL, starting from the classical Lex and YACC . Recently several modern tools have become mainstream because of their ease of use, tool support, and capabilities. We look at the main features and compare Xtext , TextX , and Antlr and determine which tool will be most appropriate for this development. Some of the deciding factors come from the availability of a formal grammar for languages that employ similar constructs that we would like to reuse or modify for our ADL . Another essential factor would be the support of programming languages for the generated parser and lexer. Some tools are centered around a specific development language, and their support for other languages may be less comprehensive. Some of the tools are tied to a particular language and IDE and have nice integration but may not be as universal as others.

8.3.1 Xtext

Xtext is a very well-established tool for building DSLs or General Purpose Languages (GPL) . It can generate a parser written in Java that the user can customize. It is heavily reliant on Java and Eclipse. It uses the Eclipse Modeling Framework (EMF) to create an Abstract Syntax Tree (AST) [128]. This allows it to efficiently do model-to-model or model-to-text transformation through the internal Ecore model in EMF . As the parser is generated Xtext creates a smart editor for the designed language in the form of another Eclipse project that is spawned from the primary Eclipse environment. The dependency on Java and Eclipse makes the tooling standard but also can be considered a hindrance to those who need to become more familiar with it or are not in favor of this family of tools.

8.3.2 TextX

TextX is based on Python and uses the Arpeggio parser. It is a tool that is not dependent on a particular IDE. The only dependency is on Python and Arpeggio. It has a very different philosophy compared to Xtext. It is a lightweight and universal tool that can fit in different environments. TextX provides fast prototyping of DSLs as it can speed up the process from

grammar generation to testing. It is easy to install as it does not have many dependencies [129]. Even if TextX is not tied to a particular IDE, it is still mainly a Python-based tool. It is also a relatively new addition to the tools in this category, which can be a problem with support from different tools, such as different editors and IDEs. This is a good solution if one is satisfied with the Python ecosystem.

8.3.3 Antlr

Antlr has been established as a DSL and GPL development tool for the last several decades. It is well supported by a number of IDEs such as Eclipse and Visual Studio with the help of plugins [130]. A unique feature of Antlr is that it can support multiple languages for the generated parsers. This is very convenient as developers can choose the language of choice for their development. Antlr is more productive than the traditional Lex and Yacc tools and is based on the Extended Backus Naur Form EBNF . The EBNF representation is based on the standard BNF which is popular in most universities' traditional compiler courses in computer science programs. There are tools that allow for grammar debugging, such as the Visual Studio Code extension, among others. Finally, numerous references and examples exist for using Antlr, as it remains an excellent approach to the creation of DSLs and GPLs.

8.4 dynADL - Dynamic Architecture Description Language

The most important part of a new design is to set the requirements that the design will need to meet. This is certainly true for an ADL . Languages that can become practical provide both simplicity and expressiveness, as well as the ability to be extended. Dependability on a particular technology for a particular language may be a good thing if the technology is well-supported and has a future. The support of tools and licensing issues are definitely a factor in their usefulness and possible adoption. All these factors express the following high-level requirements for dynADL . The development of such ADL can happen with different tools, although the analysis in the previous section leans towards the use of Antlr for the reasons mentioned above.

- The language shall be minimalistic
- Dynamic architecture definition shall be supported
- Run-time execution shall be supported
- Architectures of both systems and system of systems shall be available
- The language shall be made available as an open-source project
- The language shall allow for run-time extensions for different platforms
- The language does not try to compete with already established ADLs that specialize in certain areas
- The language shall be defined by a formal grammar in EBNF notation
- The generation of parsers from the grammar shall be supported in different programming languages
- The high-level definition of the architecture shall be separated from the platform-specific extensions

As proposed in the requirements for dynADL the main objective is to have a universal approach to defining dynamic architectures. For this to be possible, the design of dynADL separates it into a high-level language parser defined through its EBNF grammar and platform-specific extensions that can be added in addition as new platforms need to be supported. This division separates the specifics of how components are handled by different platforms, as they can be processes or threads or even sharing threads from thread pools. The platform-specific extensions support the actual handling of component loading, unloading, interface validation, and running. In this work, we focus mostly on the high-level definition of dynADL and its grammar. Examples of possible platforms are ROS, PX4, and Posix. The first implementation focuses on Posix and a partial implementation of ROS, whereas future versions can focus on PX4 and full support for ROS.

The process of designing an architecture with dynADL is shown in figure 8.1. The EBNF grammar of the language is used to produce a parser in C++. The Antlr tools also support other languages, so a Java parser is one potential choice, if desired. The created parser has listener files that define methods for each keyword in the language. This allows the user to create platform-specific extensions that can provide the actual functionality for a given platform. In the case of ROS, for example, the extension treats each component as a process and uses ROS to load, unload and run it. The interface definition in dynADL needs to be mapped to the actual interface definition of the generated system. In the case of ROS, these are the message, service and action text-based configuration files. In the case of PX4, there are message files and associated C++ code. The grammar of the dynADL language and its usage is shown in appendix B.

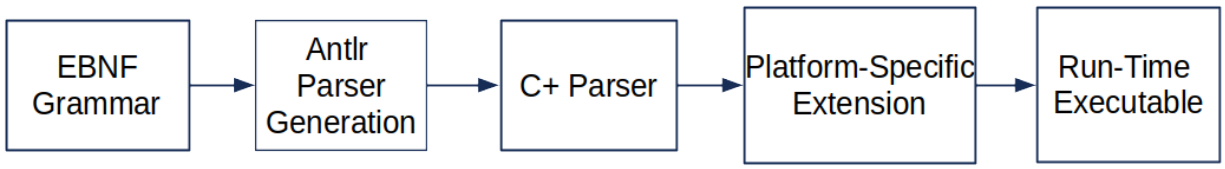

Figure 8.1. dynADL Development Process

The overall structure of the language design is shown in figure 8.2. A specification can include several dynamic packages. The figure shows package two as an example. Here a package signifies a collection of systems and can be used to represent a SoS. The description starts with listing all standard interface components, such as messages, services, actions, state

variables, and configuration parameters. This forms the standard glossary for constructing components that is used in constituent systems. The definition of each system starts with some common attributes that describe the system. What follows is the specification of each component in each system with its interface block and a definition block with general identification information. After all, systems are presented, a special section lists all possible system configurations that can be supported at run-time.

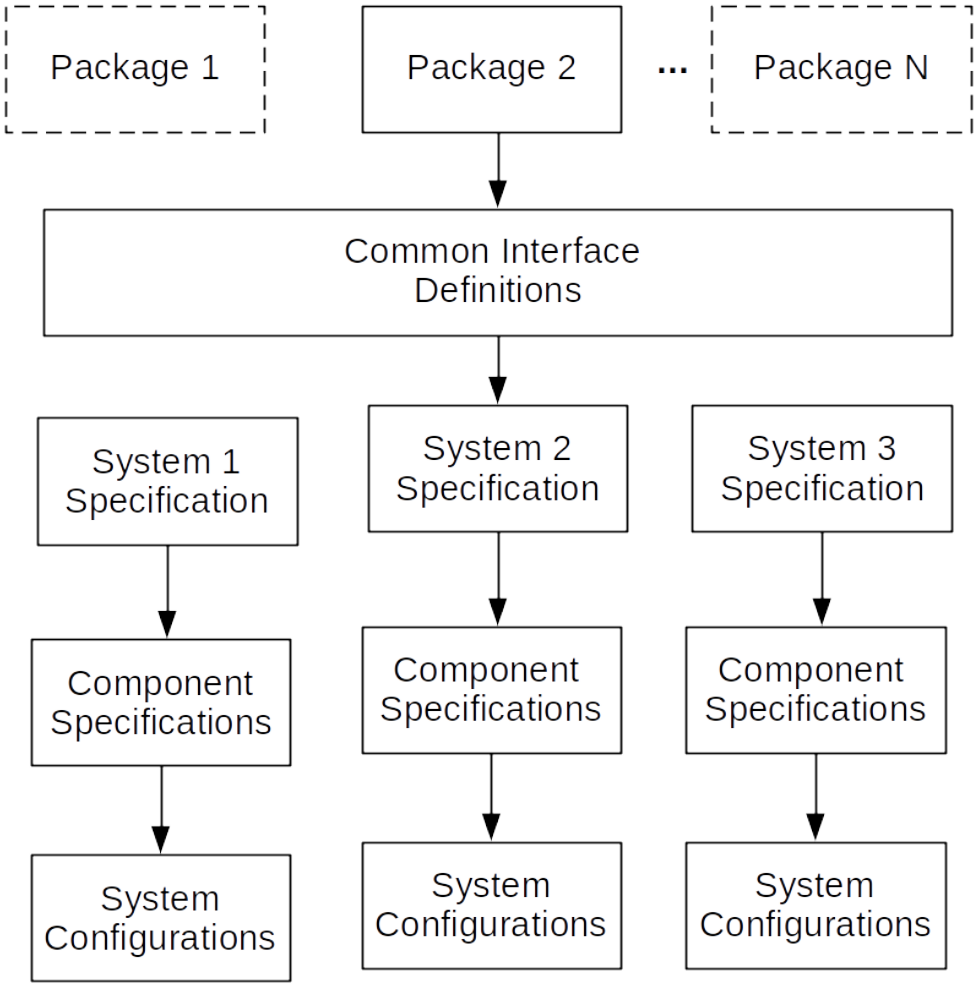

Figure 8.2. dynADL Overall Structure

dynADL allows for a detailed description of the architecture. There are two main phases in how it can be used. The first one is the generation of the final system, which includes source code and configuration files as shown in figure 8.3. After system generation is done, the user can add additional code to complete the functionality [58]. Then dynADL can execute its run-time section and start all systems with their components and perform dy-

namic reconfiguration when needed. For this, dynADL can receive commands that generate events leading to dynamic changes. In this respect, dynADL has dual nature in generating the system before run-time and then executing it during run-time . The system generation is platform-specific and different languages and configurations can be supported through platform-specific plugins.

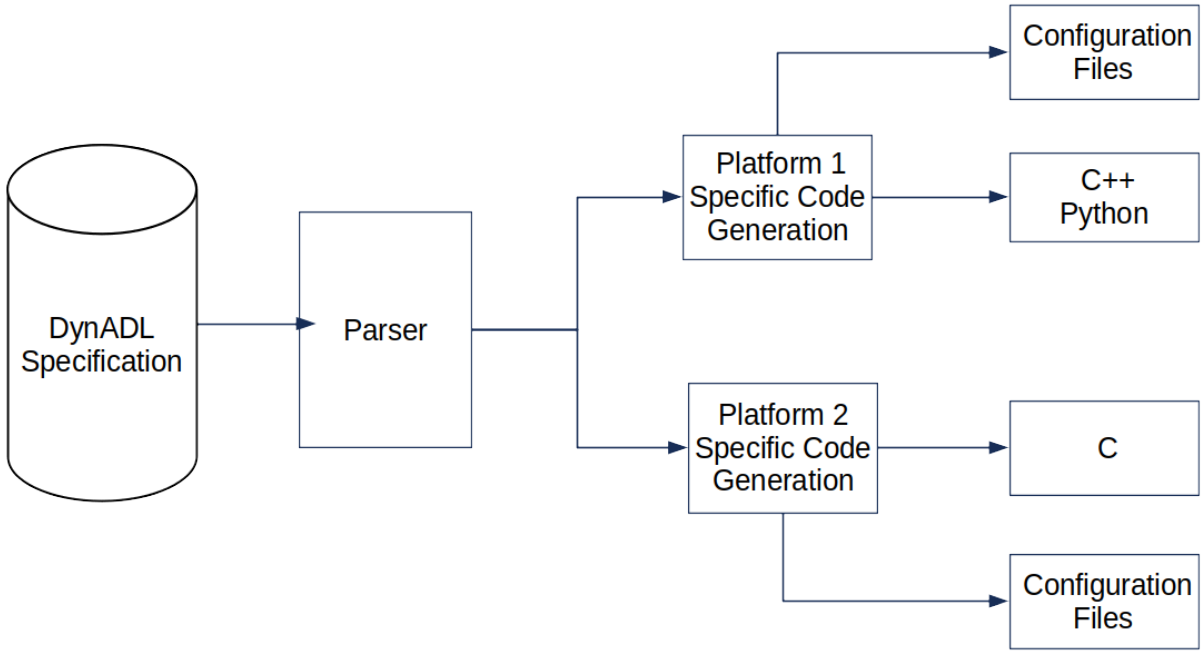

Figure 8.3. System Generation from dynADL

dynADL has been influenced by several languages, but the syntax has some similarities with the C language, although the semicolon is not necessary at the end of each line. The language has many declarative parts. The common interface definition and the system definitions are simply declaring the architecture specifics. The following run-time section is like a scripting language and defines dynamic behavior. That is why the tool can be used in two modes. The first mode generates code based on the specified architecture of the SoS.

The second mode is used to run the packages and their systems and execute the dynamic behavior in the run-time section. Both of the modes rely on platform-specific portions of the implementation.

The packages keyword in dynADL symbolizes system of systems. The system can contain multiple components that are presented with their interface sections. Each system's configurations show only the components, as each component's interface includes the connections to messages or services. This design assumes a message and service broker of some sort or a publish-subscribe mechanism. The system configurations are predefined and rely on selected and pre-approved combinations of components. The run-time section of the dynADL code allows for more arbitrary dynamic changes. This can be useful when a single component needs to be replaced during maintenance or because of some failure.

The run-time portion of dynADL has a complete set of facilities to support the management of software entities. It has support for functions that run in separate threads. Functions can also be called on the arrival of an event or through timers. Printing and logging are also part of the language. The most important part in the language are the commands. They can be run in any function or within the main function, where the execution starts. The commands allow for system configurations to be loaded and started as well as stopped and unloaded. There are commands for swapping components and saving and restoring the state of components. Commands can be run as a linear sequence or can be controlled through if and while statements that the language supports. As events and timers and all other functions run in their threads, the language supports parallelism, even if synchronization is not part of it as it is not required. C-like comments are also supported to serve as documentation.

A dynADL specification starts with the description of interfaces. Message definitions, services, actions, configuration, state, and diagnostic services represent them. Each of these can contain the basic types that are compatible with ROS and are also very similar to the types in Thrift. The data types are shown in table 8.1. Even if some of the types are not supported by Thrift, we provide a superset that is suitable for most systems. Thus the code generation for systems like ROS becomes straightforward.

Table 8.1. Supported Data Types

dynADL Data Types Compatibility		
Type	**ROS**	**Thrift**
bool	Yes	Yes
byte	Yes	Yes
char	Yes	No
int16	Yes	Yes
uint16	Yes	No
int32	Yes	yes
uint32	Yes	No
int64	Yes	Yes
uint64	Yes	No
binary	Yes	No
float32	Yes	No
float64	Yes	No
double	Yes	Yes
string	Yes	Yes

A dynADL script starts with definitions of interface components as shown in listing 8.5. The listing shows an example of each type of definition that is supported, but in a real example, there can be as many as needed. These definitions form a glossary and can be used further when the architecture is specified later. All of them use the data types provided in table 8.1.

The description of a package that contains multiple systems is shown in listing 8.6. Following are the definitions of all systems that belong to the package. Each package can have some key-value pairs to represent some features of the package. Each system can also have its own key-value pairs describing it. It can consist of an arbitrary number of components, and each component has a complete interface consisting of declarations from the interface glossary in listing 8.5.

Listing 8.5. Interface Glossary Definitions

```
message message1 {
        int32 i,
        string s
}

service service1 {
        in string str,
        out int32 i,
        out string result
}

action action1 {
        in int32 par1,
        out string s,
        update string r
}

component configuration params{
        string param1,
        int32 param2,
        float32 param3,
        double param4
}

component state state_variables {
        float32 position,
        float32 velocity,
        float32 acceleration,
        string mode
}

diagnostics service diagnostics1{
        in string result,
        out int32 r,
        out string cmd
}
```

Listing 8.6. Packages and Systems Definitions

```
package package1{
        id 1
        description "We have two systems"
        comment "package1 comment"
        // System1
        system system1
        {
                id "1"
                description "Cooling system"
                comment "system1 comment"
                component component11{
                        interface{
                                send message message1,
                                receive message message2,
                                send service service1,
                                receive service service2,
                                send action  action1,
                                send action  action2,
                                component configuration params,
                                component state state_variables,
                                diagnostics diagnostics1
                        }
                },
                component component12{
                        interface{
                                send message message1,
                                send service service1,
                                receive action  action1,
                                component configuration params,
                                component state state_variables
                        }
                }
        }
        // System 2, etc.
}
```

Listing 8.7. Functions in dynADL

```
/* Functions can be used to wait on a particular event */
function f(){
        delay 1
        print "Print from function"
        wait event event1
}

/* Functions can be called from timers */
function t(){
        print "Timer function"
}

/* Functions run in a separate thread when called with spawn */
function fsp(){
        print "Function started through spawn command"
}

function fcall(){
        print "Function to be called"
}

/* The run-time functionality starts in main */
function main(){
        spawn fsp
        call fcall
        timer periodic 1.5 t
        delay 15
}
```

The dynamic part of the script starts execution from the main function. Other functions are also supported, as shown in listing 8.7. The call operator can be used to call a function. A function can also be spawned in a separate thread through the spawn command. All timer and event functions are run in their separate threads. Each function can run commands and can contain variable declarations and if and while statements. The two operators that are supported are the increment and decrement operators. They are typically used with if and while statements.

Listing 8.8. Control Statements

```
int16 k = 10
inc k
print k
dec k
print k
if(k) then
   print "In if"
endif
int16 j = 3
while(j) do
   print j
   delay 0.1
   dec j
endwhile
int16 i = 0
if(i) then
   print "In if i"
else
   print "In else i"
endif
```

dynADL supports only if and while and variable declarations to control the execution. Everything else is accomplished through commands that provide different functionality. An example of how control statements can be used is shown in listing 8.8. The i*nc* and *dec* operators are useful when a variable needs to be incremented and decremented. Within each while and if any existing dynADL command can be invoked.

Table 8.2 shows the supported commands in dynADL.

Table 8.2. Supported Commands

dynADL Commands		
Command	**Arguments**	**Description**
run	identifier	run a component or configuration
call	identifier	calls a function
load	identifier	load a component or configuration
unload	identifier	unload a component or configuration
stop	identifier	stop a component or configuration from running
save	identifier	Save a component's state
restore	identifier	Restore a component's state
select package	identifier	Select a package as current context
select system	identifier	Select a system as current context
timer	sinle or periodic identifier	Starts a timer with a functionn call
delay	double or integer	Sleep for a certain period
print	string literal	Print a string at the console
event	identifier function	Register a functionn to be called for an event
wait event	identifier	Wait for an event to occur
exec	string literal	Executes a command in a separate process
log	string file	Logs a string to a file
spawn	identifier	Spawns a function in a thread
swap	identifier	Swaps two components

8.5 Conclusion

The idea behind the development of a dedicated dynamic ADL was to prove that this is feasible in the first place. The other goal was to explore the challenges in actual implementa-

tion and assess the possibility of it becoming a more mature tool. This exploration has been instrumental in finding complementary solutions to the plethora of design-time static modeling approaches. The effort strives to standardize the dynamic interactions and interfaces of systems and their components. The hope is that this will inspire further development in this domain. The architecture-specific extensions of dynADL can make it practical for many existing frameworks such as ROS . The availability of the grammar and the Antlr tools makes it easy to extend and customize the core language for specific applications. Adding new commands and operators can be done easily by following the example of the already-created ones.

REFERENCES

[1] J. Axelsson and A. Kobetski, "On the conceptual design of a dynamic component model for reconfigurable autosar systems," *SIGBED Rev.*, vol. 10, no. 4, pp. 45–48, Dec. 2013. DOI: 10.1145/2583687.2583698. [Online]. Available: https://doi.org/10.1145/2583687.2583698.

[2] L. Meier, D. Honegger, and M. Pollefeys, "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015, pp. 6235–6240. DOI: 10.1109/ICRA.2015.7140074.

[3] A. Kouba, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui, "Micro air vehicle link (mavlink) in a nutshell: A survey," *IEEE Access*, vol. 7, pp. 87 658–87 680, 2019. DOI: 10.1109/ACCESS.2019.2924410.

[4] A. Dubey, G. Karsai, and S. Pradhan, "Resilience at the edge in cyber-physical systems," in *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, 2017, pp. 139–146. DOI: 10.1109/FMEC.2017.7946421.

[5] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, 2003.

[6] G. Brau, J. Hugues, and N. Navet, "Towards the systematic analysis of non-functional properties in model-based engineering for real-time embedded systems," *Science of Computer Programming*, vol. 156, pp. 1–20, May 2018. DOI: 10.1016/j.scico.2017.12.007. [Online]. Available: https://oatao.univ-toulouse.fr/20731/.

[7] M. E. Shin, T. Kang, and S. Kim, "Blackboard architecture for detecting and notifying failures for component-based unmanned systems," *Journal of Intelligent & Robotic Systems*, vol. 90, no. 3, pp. 571–585, 2018.

[8] F. Campean, S. Kabir, C. Dao, Q. Zhang, and C. Eckert, "Towards a resilience assurance model for robotic autonomous systems," *Proceedings of the Design Society*, vol. 1, pp. 3189–3198, 2021. DOI: 10.1017/pds.2021.580.

[9] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, 1999. DOI: 10.1109/2.774917.

[10] A. Saadi, M. C. Oussalah, Y. Hammal, and A. Henni, "An approach for the dynamic reconfiguration of software architecture," in *2018 International Conference on Applied Smart Systems (ICASS)*, 2018, pp. 1–6. DOI: 10.1109/ICASS.2018.8651944.

[11] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming dr. frankenstein: Contract-based design for cyber-physical systems," *European journal of control*, vol. 18, no. 3, pp. 217–238, 2012.

[12] L. Kapova, B. Buhnova, A. Martens, J. Happe, and R. Reussner, "State dependence in performance evaluation of component-based software systems," in *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, ser. WOSP/SIPEW '10, San Jose, California, USA: Association for Computing Machinery, 2010, pp. 37–48, ISBN: 9781605585635. DOI: 10.1145/1712605.1712613. [Online]. Available: https://doi.org/10.1145/1712605.1712613.

[13] M. Lauer, M. Amy, J. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Resilient computing on ros using adaptive fault tolerance," *Journal of Software: Evolution and Process*, vol. 30, 2018.

[14] K. Grochowski, M. Breiter, and R. Nowak, "Serialization in object-oriented programming languages," in Aug. 2019, ISBN: 978-1-83880-333-9. DOI: 10.5772/intechopen.86917.

[15] J. Knight and E. Nguyen, "Achieving critical system survivability through software architectures," vol. 3069, Jan. 2003, pp. 51–78, ISBN: 978-3-540-23168-4. DOI: 10.1007/978-3-540-25939-8_3.

[16] M. Stoicescu, "Architecting resilient computing systems: A component-based approach," Dec. 2013.

[17] J. Wiklander, J. Eliasson, A. Kruglyak, P. Lindgren, and J. Nordlander, "Enabling component-based design for embedded real-time software," *Journal of Computers*, vol. 4, Dec. 2009. DOI: 10.4304/jcp.4.12.1309-1321.

[18] P. Hehenberger, B. Vogel-Heuser, D. Bradley, B. Eynard, T. Tomiyama, and S. Achiche, "Design, modelling, simulation and integration of cyber physical systems: Methods and applications," *Computers in Industry*, vol. 82, pp. 273–289, 2016, ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2016.05.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361516300902.

[19] D. Brugali and P. Scandurra, "Component-based robotic engineering (part i) [tutorial]," *Robotics & Automation Magazine, IEEE*, vol. 16, pp. 84–96, Jan. 2010. DOI: 10.1109/MRA.2009.934837.

[20] A. Paikan, V. Tikhanoff, G. Metta, and L. Natale, "Enhancing software module reusability using port plug-ins: An experiment with the icub robot," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2014, pp. 1555–1562.

[21] B. Y. Alkazemi, "A precise characterization of software component interfaces.," *J. Softw.*, vol. 6, no. 3, pp. 349–365, 2011.

[22] O. J. Tilak and R. R. Raje, "Temporal interaction contracts for components in a distributed system," in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, 2007, pp. 339–339. DOI: 10.1109/EDOC.2007.47.

[23] M. Stoicescu, J.-C. Fabre, and M. Roy, "Architecting resilient computing systems: A component-based approach for adaptive fault tolerance," *Journal of Systems Architecture*, vol. 73, pp. 6–16, 2017, Special Issue on Reliable Software Technologies for Dependable Distributed Systems, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2016.12.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762116302715.

[24] G. Biggs, N. Ando, and T. Kotoku, "Coordinating software components in a component-based architecture for robotics," in *SIMPAR*, 2010.

[25] J. Van, "Vulcan : Efficient component authentication and software isolation for automotive control networks," 2017.

[26] P. S. Beri and A. Mishra, "Dynamic software component authentication for autonomous systems using slack space," in *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, 2019, pp. 905–910. DOI: 10.1109/ICOEI.2019.8862570.

[27] M. Grabowski, B. Kaiser, and Y. Bai, "Systematic refinement of cps requirements using sysml, template language and contracts," in *Modellierung 2018*, I. Schaefer, D. Karagiannis, A. Vogelsang, D. Méndez, and C. Seidl, Eds., Bonn: Gesellschaft für Informatik e.V., 2018, pp. 245–260.

[28] J. Yi, D. Qi, S. H. Tan, and A. Roychoudhury, "Software change contracts," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, May 2015, ISSN: 1049-331X. DOI: 10.1145/2729973. [Online]. Available: https://doi.org/10.1145/2729973.

[29] P. Derler, E. A. Lee, M. Törngren, and S. Tripakis, "Cyber-physical system design contracts," in *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013, pp. 109–118. DOI: 10.1145/2502524.2502540.

[30] O. Ferrante, R. Passerone, A. Ferrari, L. Mangeruca, and C. Sofronis, "Bcl: A compositional contract language for embedded systems," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–6. DOI: 10.1109/ETFA.2014.7005353.

[31] S. Chaki and D. de Niz, "Contract-based verification of timing enforcers: [extended abstract]," *Ada Lett.*, vol. 36, no. 2, pp. 27–30, May 2017, ISSN: 1094-3641. DOI: 10.1145/3092893.3092898. [Online]. Available: https://doi.org/10.1145/3092893.3092898.

[32] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde, "Boosting re-use of embedded automotive applications through rich components," *Proceedings of Foundations of Interface Technologies*, vol. 2005, 2005.

[33] O. Scheickl, M. Rudorfer, and C. Ainhauser, "How timing interfaces in autosar can improve distributed development of real-time software," *INFORMATIK 2008. Beherrschbare Systeme-dank Informatik. Band 2*, 2008.

[34] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: A flexible real time scheduling framework," *Ada Lett.*, vol. XXIV, no. 4, pp. 1–8, Nov. 2004, ISSN: 1094-3641. DOI: 10.1145/1046191.1032298. [Online]. Available: https://doi.org/10.1145/1046191.1032298.

[35] R. Romagnoli, P. Griffioen, B. H. Krogh, and B. Sinopoli, "Software rejuvenation under persistent attacks in constrained environments," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 4088–4094, 2020, 21st IFAC World Congress, ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.2437. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896320331190.

[36] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, pp. 573–609, 2014.

[37] P. Baudin, F. Bobot, D. Bühler, *et al.*, "The dogged pursuit of bug-free c programs: The frama-c software analysis platform," *Commun. ACM*, vol. 64, no. 8, pp. 56–68, Jul. 2021, ISSN: 0001-0782. DOI: 10.1145/3470569. [Online]. Available: https://doi.org/10.1145/3470569.

[38] G. T. Leavens and Y. Cheon, "Design by contract with jml," 2006.

[39] S. Holthusen, S. Quinton, I. Schaefer, J. Schlatow, and M. Wegner, "Using multi-viewpoint contracts for negotiation of embedded software updates," in *PrePost@IFM*, 2016.

[40] L. Stockmann, S. Laux, and E. Bodden, "Architectural runtime verification," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 77–84. DOI: 10.1109/ICSA-C.2019.00021.

[41] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Softw. Pract. Exper.*, vol. 25, no. 7, pp. 789–810, Jul. 1995, ISSN: 0038-0644. DOI: 10.1002/spe.4380250705. [Online]. Available: https://doi.org/10.1002/spe.4380250705.

[42] M. Vai, D. Whelihan, B. Nahill, D. M. Utin, S. R. O'Melia, and R. I. Khazan, "Secure embedded systems," 2015.

[43] Z. Fang, J. Liao, and X. Zhou, "Improving system-of-systems agility through dynamic reconfiguration," in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2020, pp. 4466–4472. DOI: 10.1109/SMC42975.2020.9283236.

[44] N. Akhtar, M. M. S. Missen, N. Salamat, A. Firdous, and M. Husnain, "A study of resilient architecture for critical software-intensive system-of-systems (sisos)," *International Journal of Advanced Computer Science and Applications*, vol. 7, 2016.

[45] J. Parri, F. Patara, S. Sampietro, and E. Vicario, "A framework for model-driven engineering of resilient software-controlled systems," *Computing*, vol. 103, no. 4, pp. 589–612, Apr. 2021, ISSN: 0010-485X. DOI: 10.1007/s00607-020-00841-6. [Online]. Available: https://doi.org/10.1007/s00607-020-00841-6.

[46] D. Schneider and M. Trapp, "Runtime safety models in open systems of systems," in *Proceedings of the 2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, ser. DASC '09, USA: IEEE Computer Society, 2009, pp. 455–460, ISBN: 9780769539294. DOI: 10.1109/DASC.2009.111. [Online]. Available: https://doi.org/10.1109/DASC.2009.111.

[47] J. Pfeiffer, B. Rumpe, D. Schmalzing, and A. Wortmann, "The language of sysml v2 under the magnifying glass," *Journal of Object Technology*, vol. 21, no. 3, D. D. R. Sahar Kokaly, Ed., 3:1–15, Jul. 2022, The 18th European Conference on Modelling Foundations and Applications (ECMFA 2022), ISSN: 1660-1769. DOI: 10.5381/jot.2022.21.3.a11. [Online]. Available: http://www.jot.fm/contents/issue_2022_03/article11.html.

[48] S. E. V. and P. Samuel, "Automatic code generation from uml state chart diagrams," *IEEE Access*, vol. 7, pp. 8591–8608, 2019. DOI: 10.1109/ACCESS.2018.2890791.

[49] F. Boutekkouk and O. Fartas, "Automatic generation of sysml diagrams from vhdl code," Sep. 2015.

[50] T. Zan, H. Pacheco, and Z. Hu, "Writing bidirectional model transformations as intentional updates," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 488–491, ISBN: 9781450327688. DOI: 10.1145/2591062.2591102. [Online]. Available: https://doi.org/10.1145/2591062.2591102.

[51] A. Hristozov, E. Matson, E. Dietz, and M. Rogers, "Security of cyber-physical systems through dynamic component management," in *2023 International Journal of Engineering Research & Innovation*, IAJC, 2023.

[52] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, ISSN: 0001-0782. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[53] K. R. Apt and E.-R. Olderog, "Fifty years of hoares logic," *Form. Asp. Comput.*, vol. 31, no. 6, pp. 751–807, Dec. 2019, ISSN: 0934-5043. DOI: 10.1007/s00165-019-00501-3. [Online]. Available: https://doi.org/10.1007/s00165-019-00501-3.

[54] R. Buchanan, S. Goerger, C. Rinaudo, G. Parnell, A. Ross, and V. Sitterle, "Resilience in engineered resilient systems," *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, vol. 17, p. 154 851 291 877 790, May 2018. DOI: 10.1177/1548512918777901.

[55] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel, "A formal approach to aadl model-based software engineering," *International Journal on Software Tools for Technology Transfer*, vol. 22, pp. 219–247, 2019.

[56] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '97, Toronto, Ontario, Canada: IBM Press, 1997, p. 7.

[57] A. D. Hristozov, E. T. Matson, J. C. Gallagher, M. Rogers, and E. Dietz, "Resilient architecture framework for robotic systems," in *2022 International Conference Automatics and Informatics (ICAI)*, 2022, pp. 18–23. DOI: 10.1109/ICAI55857.2022.9960094.

[58] A. Hristozov, E. Matson, E. Dietz, and M. Rogers, "Component interface standardization in robotic systems," *Annals of Computer Science and Information Systems*, vol. 32, pp. 305–312, 2022.

[59] A. Hristozov and E. Matson, "A methodology for estimation of software architectural complexity in publish-subscribe systems," in *2022 International Conference Automatics and Informatics*, IEEE, Oct. 2022, pp. 29–34. DOI: 10.1109/ICAI55857.2022.9960011.

[60] A. Hristozov, E. Matson, J. Gallagher, E. Dietz, and M. Rogers, "Secure robotic vehicles: Vulnerabilities and mitigation strategies," in *2022 Virtual IEEE International Symposium on Technologies for Homeland Security (HST)*, IEEE, 2022, pp. 1–6.

[61] K. Sinha and O. de Weck, "Structural complexity metric for engineered complex systems and its application," in Sep. 2012, pp. 181–192, ISBN: 978-3-446-43354-0. DOI: 10.3139/9783446434127.015.

[62] J. Axelsson, "Systems-of-systems design patterns: A systematic literature review and synthesis," in *2022 17th Annual System of Systems Engineering Conference (SOSE)*, 2022, pp. 171–176. DOI: 10.1109/SOSE55472.2022.9812681.

[63] Y. Zheng, Z. Li, X. Xu, and Q. Zhao, "Dynamic defenses in cyber security: Techniques, methods and challenges," *Digital Communications and Networks*, vol. 8, no. 4, pp. 422–435, 2022, ISSN: 2352-8648. DOI: https://doi.org/10.1016/j.dcan.2021.07.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S235286482100047X.

[64] D. Thomas, "Programming with models? modeling with code. the role of models in software development.," *Journal of Object Technology*, vol. 5, pp. 15–19, Jan. 2006. DOI: 10.5381/jot.2006.5.8.c2.

[65] A. Bucchiarone, F. Ciccozzi, L. Lambers, *et al.*, "What is the future of modeling?" *IEEE Software*, vol. 38, no. 2, pp. 119–127, 2021. DOI: 10.1109/MS.2020.3041522.

[66] G. Schweiger, H. Nilsson, J. Schoeggl, W. Birk, and A. Posch, "Modeling and simulation of large-scale systems: A systematic comparison of modeling paradigms," *Appl. Math. Comput.*, vol. 365, no. C, Jan. 2020, ISSN: 0096-3003. DOI: 10.1016/j.amc.2019.124713. [Online]. Available: https://doi.org/10.1016/j.amc.2019.124713.

[67] J. Hatcliff, J. Belt, Robby, and T. Carpenter, "Hamr: An aadl multi-platform code generation toolset," in *Leveraging Applications of Formal Methods, Verification and Validation: 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 1729, 2021, Proceedings*, Rhodes, Greece: Springer-Verlag, 2021, pp. 274–295, ISBN: 978-3-030-89158-9. DOI: 10.1007/978-3-030-89159-6_18. [Online]. Available: https://doi.org/10.1007/978-3-030-89159-6_18.

[68] L. Zhang, F. Ye, K. Xie, *et al.*, "An integrated intelligent modeling and simulation language for model-based systems engineering," *Journal of Industrial Information Integration*, vol. 28, p. 100 347, 2022, ISSN: 2452-414X. DOI: https://doi.org/10.1016/j.jii.2022.100347. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2452414X2200019X.

[69] D. Lesens, "From system functional definition to software code," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[70] A. Mohsin, N. K. Janjua, S. M. S. Islam, and V. V. G. Neto, "A taxonomy of modeling approaches for systems-of-systems dynamic architectures: Overview and prospects," *ArXiv*, vol. abs/1902.09090, 2019.

[71] N. Kaminski, E. Kusmenko, and B. Rumpe, "Modeling dynamic architectures of self-adaptive cooperative systems," *J. Object Technol.*, vol. 18, 2:1–20, 2019.

[72] F. Oquendo, "Dynamic software architectures: Formally modelling structure and behaviour with pi-adl," in *2008 The Third International Conference on Software Engineering Advances*, 2008, pp. 352–359. DOI: 10.1109/ICSEA.2008.47.

[73] M. Nikolaidou, G.-D. Kapos, A. Tsadimas, V. Dalakas, and D. Anagnostopoulos, "Challenges in sysml model simulation," *Advances in Computer Science : an International Journal*, vol. 5, pp. 49–56, 2016.

[74] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications," in *Reliable Software Technologies – Ada-Europe 2009*, F. Kordon and Y. Kermarrec, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 237–250, ISBN: 978-3-642-01924-1.

[75] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev, "Automatic code generation from matlab/simulink for critical applications," in *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2014, pp. 1–6. DOI: 10.1109/CCECE.2014.6901058.

[76] M. Nikolaidou, G.-D. Kapos, A. Tsadimas, V. Dalakas, and D. Anagnostopoulos, "Challenges in sysml model simulation," *Advances in Computer Science: an International Journal*, vol. 5, no. 4, pp. 49–56, 2016.

[77] G. Agosta, E. Baldino, F. Casella, S. Cherubin, A. Leva, F. Terraneo, *et al.*, "Towards a high-performance modelica compiler," in *Proceedings of the 13th International Modelica Conference*, 2019, pp. 313–320.

[78] A. J. Kornecki and S. Johri, "Automatic code generation: Model-code semantic consistency," in *Software Engineering Research and Practice*, 2006.

[79] P. Godart, J. Gross, R. Mukherjee, and W. Ubellacker, "Generating real-time robotics control software from sysml," in *2017 IEEE Aerospace Conference*, 2017, pp. 1–11. DOI: 10.1109/AERO.2017.7943610.

[80] K. Lano and Q. Xue, "Code generation by example," in *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, INSTICC, SciTePress, 2022, pp. 84–92, ISBN: 978-989-758-550-0. DOI: 10.5220/0010973600003119.

[81] M. Funk, A. NySSen, and H. Lichter, "From uml to ansi-c - an eclipse-based code generation framework," in *ICSOFT*, 2008.

[82] M. A. Wehrmeister, "Generating ros-based software for industrial cyber-physical systems from uml/marte," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 313–320. DOI: 10.1109/ETFA46521.2020.9212077.

[83] W. Meng, J. Park, O. Sokolsky, S. Weirich, and I. Lee, "Verified generation of glue code for ros-based control systems," *Submitted for publication*, 2014.

[84] M. Elaasar, "Definition of modeling vs. programming languages," in *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I*, Limassol, Cyprus: Springer-Verlag, 2018, pp. 35–51, ISBN: 978-3-030-03417-7. DOI: 10.1007/978-3-030-03418-4_3. [Online]. Available: https://doi.org/10.1007/978-3-030-03418-4_3.

[85] E. Cavalcante, F. Oquendo, and T. Batista, "Architecture-based code generation: From π-adl architecture descriptions to implementations in the go language," in *Software Architecture*, P. Avgeriou and U. Zdun, Eds., Cham: Springer International Publishing, 2014, pp. 130–145, ISBN: 978-3-319-09970-5.

[86] R. Baduel, M. Chami, J.-M. Bruel, and I. Ober, "Sysml models verification and validation in an industrial context: Challenges and experimentation," in *Modelling Foundations and Applications*, A. Pierantonio and S. Trujillo, Eds., Cham: Springer International Publishing, 2018, pp. 132–146, ISBN: 978-3-319-92997-2.

[87] M. Hecht, J. Chen, and G. Pugliese-Rosillo, "Verification and validation of sysml models," in *2021 IEEE Aerospace Conference (50100)*, 2021, pp. 1–6. DOI: 10.1109/AERO50100.2021.9438224.

[88] R. G. Sargent, "Verification and validation of simulation models," in *Proceedings of the 37th Conference on Winter Simulation*, ser. WSC '05, Orlando, Florida: Winter Simulation Conference, 2005, pp. 130–143, ISBN: 0780395190.

[89] L. Viard, L. Ciarletta, and P.-E. Moreau, "A mission definition, verification and validation architecture," in *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 711, 2019, Revised Selected Papers, Part I*, Porto, Portugal: Springer-Verlag, 2019, pp. 281–287, ISBN: 978-3-030-54993-0. DOI: 10.1007/978-3-030-54994-7_20. [Online]. Available: https://doi.org/10.1007/978-3-030-54994-7_20.

[90] N. Weidmann, S. Salunkhe, A. Anjorin, E. Yigitbas, and G. Engels, "Automating model transformations for railway systems engineering," *J. Object Technol.*, vol. 20, 10:1–14, 2021.

[91] A. Bucchiarone and J. P. Galeotti, "Dynamic software architectures verification using dynalloy," *ECEASST*, vol. 10, Jan. 2008. DOI: 10.14279/tuj.eceasst.10.145.139.

[92] G. Regis, C. Cornejo, S. Gutiérrez Brida, *et al.*, "Dynalloy analyzer: A tool for the specification and analysis of alloy models with dynamic behaviour," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, pp. 969–973, ISBN: 9781450351058. DOI: 10.1145/3106237.3122826. [Online]. Available: https://doi.org/10.1145/3106237.3122826.

[93] J. P. Near and D. Jackson, "An imperative extension to alloy," in *Abstract State Machines, Alloy, B and Z*, M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 118–131, ISBN: 978-3-642-11811-1.

[94] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, "The electrum analyzer: Model checking relational first-order temporal specifications," ser. ASE 2018, Montpellier, France: Association for Computing Machinery, 2018, pp. 884–887, ISBN: 9781450359375. DOI: 10.1145/3238147.3240475. [Online]. Available: https://doi.org/10.1145/3238147.3240475.

[95] Esmaeilsabzali, Shahram, Day, Nancy A., and Serna, Jose, "Dash: Declarative modelling with control state hierarchy (preliminary version)," Tech. Rep., 2018. [Online]. Available: http://hdl.handle.net/10012/16037.

[96] A. M. Law and D. M. Kelton, *Simulation Modeling and Analysis*, 3rd. McGraw-Hill Higher Education, 2015, ISBN: 0070592926.

[97] W. A. Menner, "Introduction to modeling and simulation," 2015.

[98] J.-F. Tilman, R. Sezestre, and A. Schyn, "Simulation of system architectures with aadl," in *Embedded Real Time Software and Systems (ERTS2008)*, Toulouse, France, 2008, ISBN: insu-02269764.

[99] P. Dissaux and O. Marc, "Executable aadl real time simulation of aadl models," *CEUR Workshop Proceedings*, vol. 1233, Jan. 2014.

[100] J. Liu, T. Li, Z. Ding, Y. Qian, H. Sun, and J. He, "Aadl+: A simulation-based methodology for cyber-physical systems," *Frontiers of Computer Science*, vol. 13, pp. 516–538, 2018.

[101] G.-D. Kapos, A. Tsadimas, C. Kotronis, V. Dalakas, M. Nikolaidou, and D. Anagnostopoulos, "A declarative approach for transforming sysml models to executable simulation models," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 51, no. 6, pp. 3330–3345, 2021. DOI: 10.1109/TSMC.2019.2922153.

[102] D. C. Café, F. V. dos Santos, C. Hardebolle, C. Jacquet, and F. Boulanger, "Multi-paradigm semantics for simulating sysml models using systemc-ams," in *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, 2013, pp. 1–8.

[103] E. Seidewitz., "On a metasemantic protocol for modeling language extension," in *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD,*, INSTICC, SciTePress, 2020, pp. 465–472, ISBN: 978-989-758-400-8. DOI: 10.5220/0009181604650472.

[104] A. Berriche, F. Mhenni, A. Mlika, and J.-Y. Choley, "Towards model synchronization for consistency management of mechatronic systems," *Applied Sciences*, vol. 10, no. 10, 2020, ISSN: 2076-3417. DOI: 10.3390/app10103577. [Online]. Available: https://www.mdpi.com/2076-3417/10/10/3577.

[105] H. Li, C. Zhan, H. Wu, M. Yu, J. Dai, and W. Zou, "Architecting commercial aircraft with a domain specific language extended from sysml," *Journal of Physics: Conference Series*, vol. 1827, no. 1, p. 012 100, Mar. 2021. DOI: 10.1088/1742-6596/1827/1/012100. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1827/1/012100.

[106] A. Sutton and J. Maletic, "Mappings for accurately reverse engineering uml class models from c++," in *12th Working Conference on Reverse Engineering (WCRE'05)*, 2005, 10 pp.–184. DOI: 10.1109/WCRE.2005.21.

[107] V. de Oliveira Neves, A. Bertolino, G. de Angelis, and L. Garcés, "Do we need new strategies for testing systems-of-systems?" In *2018 IEEE/ACM 6th International Workshop on Software Engineering for Systems-of-Systems (SESoS)*, 2018, pp. 29–32.

[108] N. Messe, N. Belloir, V. Chiprianov, I. Cherfa, R. Fleurquin, and S. Sadou, "Development of secure system of systems needing a rapid deployment," in *2019 14th Annual Conference System of Systems Engineering (SoSE)*, 2019, pp. 152–157. DOI: 10.1109/SYSOSE.2019.8753857.

[109] C. E. Dridi, Z. Benzadri, and F. Belala, "System of systems engineering: Meta-modelling perspective," in *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, 2020, pp. 000 135–000 144. DOI: 10.1109/SoSE50414.2020.9130465.

[110] J. Criado, L. Iribarne, and N. Padilla, "Heuristics-based mediation for building smart architectures at run-time," *Computer Standards and Interfaces*, vol. 75, p. 103 501, 2021, ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2020.103501. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548920303883.

[111] F. Petitdemange, I. Borne, and J. Buisson, "Modeling system of systems configurations," in *2018 13th Annual Conference on System of Systems Engineering (SoSE)*, 2018, pp. 392–399. DOI: 10.1109/SYSOSE.2018.8428737.

[112] A. Hristozov and E. Matson, "Modeling aspects of dynamically reconfigurable system of systems," in *Conference on Systems Engineering Research - CSER 2023*, IEEE, 2023.

[113] M. Hasanagic, T. Fabbri, P. G. Larsen, V. Bandur, P. W. V. Tran-Jørgensen, and J. Ouy, "Code generation for distributed embedded systems with vdm-rt," *Des. Autom. Embed. Syst.*, vol. 23, pp. 153–177, 2019.

[114] N. Nguyen, F. Mhenni, and J.-Y. Choley, "Altarica 3.0 code generation from sysml models," in Jun. 2018, pp. 2435–2440, ISBN: 9781351174664. DOI: 10.1201/9781351174664-306.

[115] H. Derhamy, J. Eliasson, and J. Delsing, "System of system composition based on decentralized service-oriented architecture," *IEEE Systems Journal*, vol. 13, no. 4, pp. 3675–3686, 2019. DOI: 10.1109/JSYST.2019.2894649.

[116] F. Belala, N. Hameurlain, and A. Seghiri, "Modeling the Dynamic Reconfiguration in Smart Crisis Response Systems," in *17th International Conference on Evaluation of Novel Approaches to Software Engineering*, Online Streaming, Portugal: SCITEPRESS - Science and Technology Publications, Apr. 2022, pp. 162–173. DOI: 10.5220/0011069300003176. [Online]. Available: https://hal-univ-pau.archives-ouvertes.fr/hal-03658634.

[117] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall, "Metacsl: Specification and verification of high-level properties," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds., Cham: Springer International Publishing, 2019, pp. 358–364, ISBN: 978-3-030-17462-0.

[118] S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet, "A formal toolchain for offline and run-time verification of robotic systems," *Robotics and Autonomous Systems*, 2022. [Online]. Available: https://hal.laas.fr/hal-03683044.

[119] D. Bjørner and K. Havelund, "40 years of formal methods - some obstacles and some possibilities?" In *FM*, 2014.

[120] D. Schneider and M. Trapp, "B-space: Dynamic management and assurance of open systems of systems," *Journal of Internet Services and Applications*, vol. 9, Dec. 2018. DOI: 10.1186/s13174-018-0084-5.

[121] K. Hobbs, M. L. Mote, M. Abate, S. D. Coogan, and E. Feron, "Run time assurance for safety-critical systems: An introduction to safety filtering approaches for complex control systems," *ArXiv*, vol. abs/2110.03506, 2021.

[122] E. Silva, T. Batista, and F. Oquendo, "A mission-oriented approach for designing system-of-systems," in *2015 10th System of Systems Engineering Conference (SoSE)*, 2015, pp. 346–351. DOI: 10.1109/SYSOSE.2015.7151951.

[123] R. Giachetti, S. Wangert, and R. Eldred, "Interoperability analysis method for mission-oriented system of systems engineering," in *2019 IEEE International Systems Conference (SysCon)*, 2019, pp. 1–6. DOI: 10.1109/SYSCON.2019.8836808.

[124] M. Himsolt, "Gml: A portable graph file format," Technical report, Universitat Passau, Tech. Rep., 1997.

[125] W. Abernethy, *Programmer's Guide to Apache Thrift*. Simon and Schuster, 2019.

[126] P. Feiler, D. Gluch, and J. Hudak, "The architecture analysis & design language (aadl): An introduction," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2006-TN-011, 2006. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7879.

[127] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds., Cambridge University Press, 2000.

[128] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," ser. OOPSLA '10, Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 307–309, ISBN: 9781450302401. DOI: 10.1145/1869542.1869625. [Online]. Available: https://doi.org/10.1145/1869542.1869625.

[129] I. Dejanovi, R. Vaderna, G. Milosavljevi, and . Vukovi, "Textx: A python tool for domain-specific languages implementation," *Knowledge-Based Systems*, vol. 115, pp. 1–4, 2017, ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2016.10.023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705116304178.

[130] F. Ortin, J. Quiroga, O. Rodriguez-Prieto, and M. Rodríguez, "An empirical evaluation of lex/yacc and antlr parser generation tools," *PLOS ONE*, vol. 17, e0264326, Mar. 2022. DOI: 10.1371/journal.pone.0264326.

A. DEVELOPMENT SETUP

A.1 Dependencies and Makefile

The build procedure requires the installation of a relatively recent version of Antlr , for example 4.7.2 or newer. A Makefile manages the build with *make* and *make clean* commands. Part of the contents of the Makefile are shown in listing A.1. The assumptions are that Antlr and Java are installed. Another prerequisite is to have a C++ compiler since the generated code is chosen to be C++.

A.2 IDE

There are several IDEs that support Antlr development. Among them are Eclipse, VS Code, Netbeans and IneliJ. The first three of the IDEs are open-source and have pretty good functionality. For this effort VS Code was selected because of its ease of use and the nice Antlr extension. Visual Studio Code is a platform that can allow for development of C++ projects in a very elegant way. It comes with .json files that allow for flexible configuration. The tasks.json file allows for performing build from the IDE. The easiest way to do this from VS Code is to call the external make file for building and make clean for cleanup. The tasks.json file is included in listing A.2. Similarly the launch.json is used for the run and debug command and for the project there are two configurations: one for the C++ project and one for the grammar file as shown in listing A.3. This approach allows for development of the grammar and the parser code within the same IDE configuration.

A.3 Debugging

The two configurations for running and debugging allow to separately develop the grammar and to work with the parse tree. The code generated from the grammar can be debugged through the other configuration in the launch.json file as a usual C++ project. Since VS Code is working on multiple platforms it can use different debuggers and compilers. In Linux this can be gcc and gdb or clang. The VS Code Antlr extension allows for breakpoints to be put in the EBNF grammar which is extremely helpful since the development of the grammar

Listing A.1. Makefile

```
OUTPUT=output
GENERATED=generated
RUNTIME=/usr/local
ANTLRJARDIR=/home/anton/ANTLR-VER
CCARGS=-c -I $(RUNTIME)/include/antlr4-runtime/
  -I $(GENERATED)  -std=c++11 -g
LDARGS=-g
LIBS=$(RUNTIME)/lib/libantlr4-runtime.a
JAVA=/usr/bin/java
CC=g++
GRAMMAR=dynadl
ANTLR4=$(JAVA) -jar $(ANTLRJARDIR)/antlr-4.9.3-complete.jar

ANTLRGEN=BaseListener Lexer Listener Parser
OBJS=$(addsuffix .o,$(addprefix $(OUTPUT)/$(GRAMMAR),
       $(ANTLRGEN)))
GSOURCES=$(addsuffix .cpp,$(addprefix
          $(GENERATED)/$(GRAMMAR),$(ANTLRGEN)))

.precious: $(GSOURCES)

all: dynadl

dynadl: dirs antlr4 dynadl.cpp dynadlWalkListener.cpp $(OBJS)
$(CC) $(CCARGS) dynadl.cpp -o $(OUTPUT)/dynadl.o
$(CC) $(CCARGS) dynadlWalkListener.cpp -o
                $(OUTPUT)/dynadlWalkListener.o
$(CC) $(LDARGS) $(OUTPUT)/dynadl.o $(OUTPUT)/dynadlWalkListener.o
                $(OBJS) $(LIBS) -o dynadl

antlr4: $(GENERATED)/.generated;

$(GENERATED)/.generated: $(GRAMMAR).g4
$(ANTLR4) -Dlanguage=Cpp -o $(GENERATED) $(GRAMMAR).g4
@touch $(GENERATED)/.generated

$(OUTPUT)/%.o : $(GENERATED)/%.cpp
$(CC) $(CCARGS) $< -o $@

$(GENERATED)/%.cpp: $(GENERATED)/.generated;

dirs:; mkdir -p $(OUTPUT) $(GENERATED)
clean:; rm -rf dynadl Debug $(OUTPUT) $(GENERATED)
               *.msg *.action *.srv *.cfg *.sta
```

Listing A.2. tasks.json

```
{
        "tasks": [
        {
                "type": "cppbuild",
                "label": "make",
                "command": "make",
                "args": [],
                "options": {
                        "cwd": "${fileDirname}"
                },
                "problemMatcher": [
                "$gcc"
                ],
                "group": "build",
                "detail": "Task generated by Debugger."
        },
        {
                "type": "cppbuild",
                "label": "C/C++ make clean",
                "command": "make",
                "args": [
                "clean"
                ],
                "options": {
                        "cwd": "${fileDirname}"
                },
                "problemMatcher": [
                "$gcc"
                ],
                "group": "build",
                "detail": "Task generated by Debugger."
        }
        ],
        "version": "2.0.0"
}
```

Listing A.3. launch.json

```
{
        "version": "0.2.0",
        "configurations": [
        {
                "name": "g++-9 - Build and debug active file",
                "type": "cppdbg",
                "request": "launch",
                "program": "${fileDirname}/${fileBasenameNoExtension}",
                "args": ["sample.expr"],
                "stopAtEntry": false,
                "cwd": "${fileDirname}",
                "environment": [],
                "externalConsole": false,
                "MIMode": "gdb",
                "setupCommands": [
                {
                        "description": "Enable pretty-printing for gdb",
                        "text": "-enable-pretty-printing",
                        "ignoreFailures": true
                },
                {
                        "description": "Set Disassembly Flavor to Intel",
                        "text": "-gdb-set disassembly-flavor intel",
                        "ignoreFailures": true
                }
                ],
                "preLaunchTask": "C/C++: g++-9 build active file",
                "miDebuggerPath": "/usr/bin/gdb"
        },
        {
                "name": "Debug ANTLR4 grammar",
                "type": "antlr-debug",
                "request": "launch",
                "input": "sample.expr",
                "grammar": "Thrift.g4",
                //"actionFile": "grammars/exampleActions.js",
                "startRule": "main",
                "printParseTree": true,
                "visualParseTree": true
        }

        ]
}
```

may be even more challenging than the development of the parser extensions in the chosen programming language.

A.4 Running

Running of the generated executable can happen from the command line. It consists of two steps: code generation and runtime sequence. The command line interface of the dynADL is simple. Examples of a typical command line sequence follow:

```
dynadl sample.script --generate posix
dynadl sample.script --runtime posix
```

The executable can also be run in VS Code. VS Code uses the launch.json file to run the two different targets: the grammar itself and the executable parser, generated from the grammar.

B. DYNADL

B.1 Command Line Interface

The executable for dynADL is run in a command shell. The implementation was developed on Linux, but because of the generic nature of Antlr, the code can be used in other operating systems. The command syntax is shown in a general form:

```
./dynadl script_file.txt [--generate | --runtime] [posix | ros | px4]
```

In the case of providing the *generate* argument, dynadl will generate configuration files and source code for the provided platform. When *runtime* is selected, then the dynamic commands from the run-time section are executed for the selected platform. In both cases, the process starts with parsing the dynADL code and ensuring it complies with the language's grammar rules. Depending on the arguments, different behavior follows. The run-time section of the script always has a main function where the execution starts. Other functions can be invoked from there.

B.2 Architecture Description Sample

The general description of a large-scale system can be accomplished by defining each SoS as a package containing multiple systems. There can be multiple packages that allow for aggregating several SoS in a bigger SoS. A general example is shown in listings B.1 and B.2.

B.3 Code Generation

The generation of ROS focuses on the interfaces of components. The generation creates a directory structure as shown in listing B.3. Each component has a separate subfolder for messages, services, actions, configuration, state, and diagnostics. The generated files for messages, services, and actions comply to the format for ROS2 as shown in listing B.4. The config state and diagnostics files are generated in a similar fashion to complement the standard ROS files.

The code generation for Posix follows a different approach. The files contain JSON format for all portions of the component interface. This enables the user to parse the

Listing B.1. General Architecture Description

```
package package1{
        // Package key-value pairs
        system system1
        {
                component component11{
                        interface{
                                // Interface contents
                        }
                },
                component component12{
                        interface{
                                // Interface contents
                        }
                },
                component component13{
                        interface{
                                // Interface contents
                        }
                },
                system configuration sys_config1{
                        // system configuration contents
                },
                system configuration sys_config2{
                        // system configuration contents
                },
        }, // end system1
```

Listing B.2. General Architecture Description (cont.)

```
        system system2
        {
                // System key-value pairs
                component component21{
                        interface{
                        }
                },
                component component22{
                        interface{
                        }
                },
                component component23{
                        interface{
                        }
                },
                system configuration sys_config3{
                        // System configuration contents
                },
                system configuration sys_config4{
                        // System configuration contents
                },
        }, // end system2

        // More systems

}, // end package1
package package2{

        // Systems of package2

        ...
        }
```

Listing B.3. Generated Directory Structure for ROS

```
.
|---component11
|    |---action
|    |    |---action1.action
|    |    |---action1.action.json
|    |---component11.cpp
|    |-- config
|    |    |---params.cfg.json
|    |---config.cpp
|    |---datatypes.h
|    |---diag
|    |    |---diagnostics1.diag.json
|    |---diagnostics.cpp
|    |---diagnostics.h
|    |---Makefile
|    |---msg
|    |    |---message1.msg
|    |    |---message1.msg.json
|    |---state
|    |    |---state_variables.sta.json
|    |---state.cpp
|    |---state.h
|---component12
|    |---component12
|    |---component12.cpp
|    |---config
|    |    |-- params.cfg.json
|    |---config.cpp
|    |---datatypes.h
|    |---diag
|    |---diagnostics.cpp
|    |---diagnostics.h
|    |---Makefile
|    |---msg
|    |    |---message1.msg
|    |    |---message1.msg.json
|    |---srv
|    |    |-- service1.srv
|    |    |---service1.srv.json
|    |---state
|    |    |---state_variables.sta.json
|    |---state.cpp
|    |---state.h
```

Listing B.4. Generated Files Contents for ROS

```
file message1.msg
string str
int16 i

file service1.srv
string str
---
int32 i
string result

file action1.action
int32 par1
---
string s
---
string r

file params.cfg
string param1
int32 param2
float32 param3
double param4

file state_variables.sta
float32 position
float32 velocity
float32 acceleration
string mode
```

interface and process it easier. A Posix implementation relies on standard implementations available on Posix-compliant OSes. In addition to the interface files, some C++ template code is also produced. The C++ code takes care of the diagnostics interface, the component's configuration, and the state handling. This approach automates the procedure of interface and boilerplate code generation to speed up development.

B.4 EBNF Grammar

The grammar is shown in the following listings in EBNF format used to generate a C++ parser through the Antlr tool. EBNF is an extension of the original BNF grammar definition language. The existence of a formal grammar allows for easier extensions in the future. The same EBNF grammar can be used to generate parsers in different programming languages as Antlr supports many. The default language for parser code generation in Antlr is Java, althoug C++ was used to develop dynADL.

Listing B.5. Generated Files Contents for Posix

```
file message1.msg
{
        "id": "message1",
        "var0": [
        "string",
        "str"
        ],
        "var1": [
        "int16",
        "i"
        ]
}

file action1.action
{
        "id": "action1",
        "in": {
                "var0": [
                "int32",
                "par1"
                ]
        },
        "out": {
                "var1": [
                "string",
                "s"
                ]
        },
        "update": {
                "var2": [
                "string",
                "r"
                ]
        }
}
```

Listing B.6. Generated header files

```
#ifndef DATATYPES_H
#define DATATYPES_H

typedef int int16;
typedef unsigned int uint16;
typedef int int32;
typedef unsigned int uint32;
typedef int int64;
typedef unsigned int uint64;
typedef float float32;
typedef double float64;
typedef char byte;
typedef char binary;

struct MyStruct{
        int id;
        char cmd[20];
};
#endif

#ifndef STATE_H
#define STATE_H

void save(void);
void restore(void);

#endif

#ifndef DIAGNOSTICS_H
#define DIAGNOSTICS_H

void d_unload(void);
void d_start(void);
void d_stop(void);
void d_save(void);
void d_restore(void);
bool get_start(void);

void diag(std::string component_id);

#endif
```

Listing B.7. Generated main file contents sample for each component

```
//Compile: g++ -std=c++11 -pthread
//              component11.cpp -o component11

#include <iostream>
#include <thread>
#include <unistd.h>
#include "state.h"
#include "diagnostics.h"
using namespace std;



void config(void);
void state(void);

int main(int argc, char *argv[])
{
        cout << "Starting component11"<< endl;
        config();
        thread th1(diag,argv[0]);
        for(;;){
                sleep(1);
        }
        return 0;
}
```

Listing B.8. Generated configuration file contents

```
#include <iostream>
#include <unistd.h>
#include "datatypes.h"
using namespace std;

struct params{
        string param1 = "str";
        int32 param2 = 10;
        float32 param3 = 2.71;
        double param4 = 3.14;
}cfg_struct;

params* get_config_struct(void){
        return &cfg_struct;
}

void config(void)
{
        cout << "Configuration handling routine" << endl;
        cout << "Configuraton parameters:" << endl;
        cout << cfg_struct.param1 << endl;
        cout << cfg_struct.param2 << endl;
        cout << cfg_struct.param3 << endl;
        cout << cfg_struct.param4 << endl;
}
```

Listing B.9. Generated diagnostics file contents

```
void diag(string component_id)
{
        diagnostics1_in diag_struct_in;
        diagnostics1_out diag_struct_out;
        MyStruct mystruct;
        cout << "Diagnostics routine for component: "
             << component_id << endl;
        try{
                //Open a message queue.
                message_queue mq
                (open_or_create
                ,"component11"
                ,10
                ,sizeof(MyStruct)
                );
                unsigned int priority;
                message_queue::size_type recvd_size;
                //Receive some messages
                for(;;){
                        std::cout << "Blocking to read" << std::endl;
                        mq.receive(&mystruct, sizeof(MyStruct),
                         recvd_size, priority);
                        if(recvd_size != sizeof(MyStruct)){
                                std::cout <<
                                "Received unexpected length"
                                << std::endl;
                                continue;
                        }
                        std::cout << mystruct.id << std::endl;
                        std::cout << mystruct.cmd << std::endl;
                        if(!strcmp(mystruct.cmd,"start")){
                                d_start();
                        } // ... More commands follow,
                }
        }
        catch(interprocess_exception &ex){
                message_queue::remove("send_queue");
                std::cout << ex.what() << std::endl;
                return;
        }
        message_queue::remove("send_queue");
        return;
}
#include <boost/interprocess/detail/config_end.hpp>
```

Listing B.10. Generated state file contents

```
struct state_variables{
        float32 position;
        float32 velocity;
        float32 acceleration;
        string mode;
}state_struct;

auto ser_file_name = "ser_component11.bin";
std::fstream * s_ptr;
bitsery::Serializer<bitsery::OutputBufferedStreamAdapter> * ser_ptr;
bitsery::Deserializer<bitsery::InputStreamAdapter> *des_ptr;

void save(void)
{
        s_ptr = new std::fstream {ser_file_name, s_ptr->binary |
                s_ptr->trunc | s_ptr->out};
        ser_ptr = new bitsery::Serializer
                  <bitsery::OutputBufferedStreamAdapter>(*s_ptr);
        ser_ptr->value4b(state_struct.position);
        ser_ptr->value4b(state_struct.velocity);
        ser_ptr->value4b(state_struct.acceleration);
        ser_ptr->text1b(state_struct.mode,state_struct.mode.length());
        ser_ptr->adapter().flush();
        s_ptr->close();
        delete(ser_ptr);
        delete(s_ptr);
}

void restore(void)
{
        s_ptr = new std::fstream {ser_file_name, s_ptr->binary
                          | s_ptr->in};
        des_ptr = new bitsery::Deserializer
                  <bitsery::InputStreamAdapter>(*s_ptr);
        des_ptr->value4b(state_struct.position);
        des_ptr->value4b(state_struct.velocity);
        des_ptr->value4b(state_struct.acceleration);
        des_ptr->text1b(state_struct.mode,state_struct.mode.length());
        s_ptr->close();
        delete(des_ptr);
        delete(s_ptr);
}

void state(void)
{
        cout << "State handling routine" << endl;
}
```

Listing B.11. Antlr grammar for dynADL

```
grammar dynadl;

prog
: packages
;

packages
: ML_COMMENT* SL_COMMENT* definition* pckg*
 (function_definition)*main_function_definition EOF
;

script
: (commands | statement)*
;

commands
:  (start_command | load_command | unload_command |
 stop_command | save_command | restore_command |
 select_package_command |select_system_command |
 swap_command | delay_command | event_command |
 call_command | print_command| timer_command |
 exec_command | log_command | spawn_command |
 wait_command);

block
: (commands | dec_statement | inc_statement)*
;

pckg
: PACKAGE IDENTIFIER '{' kv+ system* system '}'
| COMMA
;

system
: SYSTEM IDENTIFIER '{' kv+ component* component
        system_configuration* '}' | COMMA
;

component
: 'component' IDENTIFIER '{' iface* iface '}'
| COMMA
;
```

Listing B.12. Antlr grammar for dynADL(continued)

```
system_configuration
: 'system' 'configuration' IDENTIFIER
'{' component_decl* '}' | COMMA
;

iface
: 'interface'
  '{' declaration*  declaration '}'
;

declaration
: variable_declaration | message_decl | service_decl |
action_decl | component_configuration_decl  |
component_state_decl | diagnostics_decl | COMMA
;

message_decl
:  SEND_RECEIVE  'message' IDENTIFIER
;

service_decl
: SEND_RECEIVE  'service' IDENTIFIER
;

action_decl
: SEND_RECEIVE  'action' IDENTIFIER
;

component_state_decl
: 'component' 'state' IDENTIFIER
;

component_configuration_decl
: 'component' 'configuration' IDENTIFIER
;

diagnostics_decl
: DIAGNOSTICS IDENTIFIER
;

component_decl
: 'component' IDENTIFIER
;
```

Listing B.13. Antlr grammar for dynADL(continued)

```
definition
: variable_declaration_statement | message | service |
action | component_configuration | component_state | COMMA
;

message
: 'message' IDENTIFIER '{' field* '}' type_annotations?
;

component_state
: 'component' 'state' IDENTIFIER '{' field* '}'
type_annotations?
;

component_configuration
: 'component' 'configuration' IDENTIFIER '{' cfg_field* '}'
type_annotations?
;

service
: DIAGNOSTICS? 'service' IDENTIFIER '{' (IN_OUT field)* '}'
type_annotations?
;

action
: 'action' IDENTIFIER '{' ((IN_OUT | UPDATE) field)* '}'
type_annotations?
;

field
: field_id? field_type IDENTIFIER ('=' const_value)?
type_annotations? list_separator?
;

field_id
: integer ':'
;

cfg_field
: field_type IDENTIFIER '=' value list_separator?
;

function_definition
: 'function' IDENTIFIER '(' ')' '{' script '}'
;
```

Listing B.14. Antlr grammar for dynADL(continued)

```
main_function_definition
: 'function' 'main' '(' ')' '{' script '}'
;

type_annotations
: '(' type_annotation* ')'
;

type_annotation
: IDENTIFIER ('=' annotation_value)? list_separator?
;

annotation_value
: dbl | integer | LITERAL
;

field_type
: base_type | IDENTIFIER | container_type
;

base_type
: real_base_type type_annotations?
;

container_type
: type_annotations?
;

const_value
: integer | dbl | LITERAL | IDENTIFIER
;

integer
: INTEGER | HEX_INTEGER
;

INTEGER
: ('+' | '-')? DIGIT+
;

HEX_INTEGER
: '-'? '0x' HEX_DIGIT+
;
```

Listing B.15. Antlr grammar for dynADL(continued)

```
dbl
: DOUBLE
;

DOUBLE
: ('+' | '-')? ( DIGIT+ ('.' DIGIT+)? | '.' DIGIT+ )
(('E' | 'e') INTEGER)?
;

list_separator
: COMMA | ';'
;

real_base_type
:  TYPE_BOOL | TYPE_BYTE | TYPE_CHAR | TYPE_INT16 |
 TYPE_UINT16 | TYPE_INT32| TYPE_UINT32 | TYPE_INT64
 | TYPE_UINT64 | TYPE_FLOAT32 | TYPE_FLOAT64 |
 TYPE_DOUBLE | TYPE_STRING | TYPE_BINARY
;

start_command
: 'start' IDENTIFIER
;

load_command
: 'load' IDENTIFIER
;

unload_command
: 'unload' IDENTIFIER
;

stop_command
: 'stop' IDENTIFIER
;

save_command
: 'save' IDENTIFIER
;

restore_command
: 'restore' IDENTIFIER
;
```

Listing B.16. Antlr grammar for dynADL(continued)

```
select_package_command
: 'select' PACKAGE IDENTIFIER
;

select_system_command
: 'select' SYSTEM IDENTIFIER
;

delay_command
: 'delay' (dbl | integer)
;

timer_command
: 'timer' (SINGLE_PERIODIC) (dbl | integer) IDENTIFIER
;

print_command
: 'print' (STRINGLITERAL | IDENTIFIER)
;

event_command
: 'event' IDENTIFIER
;

wait_command
: 'wait' IDENTIFIER IDENTIFIER ('('')')?
;


exec_command
: 'exec' STRINGLITERAL
;

log_command
: 'log' STRINGLITERAL STRINGLITERAL
;

spawn_command
: 'spawn' IDENTIFIER ('('')')?
;
```

Listing B.17. Antlr grammar for dynADL(continued)

```
event_parameter_list
: '(' ( event_parameter (',' event_parameter)* )? ')'
;

event_parameter
: real_base_type IDENTIFIER?
;

expression
: basic_expression
;

basic_expression
: value
| IDENTIFIER
| real_base_type
;

call_command
: 'call' IDENTIFIER ('('')')?
;

statement
: if_statement
| while_statement
| inc_statement
| dec_statement
| basic_statement
;

while_statement
: 'while' '(' expression ')' DO block ENDWHILE ;

if_statement
: 'if' '(' expression ')' THEN block (ELSE block)? ENDIF
;

inc_statement
: 'inc' IDENTIFIER
;
```

Listing B.18. Antlr grammar for dynADL(continued)

```
dec_statement
: 'dec' IDENTIFIER
;

basic_statement
: ( variable_declaration_statement | expression )
;

variable_declaration_statement
: ( identifier_list | variable_declaration |
'(' variable_declaration_list ')' ) ( '=' expression )?
;

variable_declaration_list
: variable_declaration? (',' variable_declaration? )* ;

variable_declaration
: real_base_type  IDENTIFIER
;

identifier_list
: '(' ( IDENTIFIER? ',' )* IDENTIFIER? ')'
;

swap_command
: 'swap' IDENTIFIER IDENTIFIER ((AT (dbl | integer)) | ONCE)
;

key_type
:  KEY_ID | KEY_DESC | KEY_COMMENT
;

list_kv
: '[' kv + ']'
;

kv
: key value
;
```

Listing B.19. Antlr grammar for dynADL(continued)

```
value
: integer
| realnum
| dbl
| stringliteral
| list_kv
;

key
: key_type
;

realnum
: REAL
;

stringliteral
: STRINGLITERAL
;

STRINGLITERAL
: '"' ~ '"'* '"'
;

REAL
: SIGN? DIGIT* '.' DIGIT + MANTISSA?
;

SIGN
: '+' | '-'
;

MANTISSA
: 'E' SIGN DIGIT
;

KEY_ID: 'id';
KEY_DESC: 'description';
KEY_COMMENT: 'comment';

SEND_RECEIVE
: 'send' | 'receive'
;
```

Listing B.20. Antlr grammar for dynADL(continued)

```
IN_OUT
: 'in' | 'out'
;

UPDATE
: 'update'
;

DIAGNOSTICS
: 'diagnostics'
;

PACKAGE
: 'package'
;

SYSTEM
: 'system'
;

AT
: 'at'
;

ONCE
: 'once'
;

SINGLE_PERIODIC
: 'single' | 'periodic'
;

DO
: 'do'
;

ENDWHILE
: 'endwhile'
;

ENDIF
: 'endif'
;
```

Listing B.21. Antlr grammar for dynADL(continued)

```
THEN
: 'then'
;

ELSE
: 'else'
;

TYPE_BOOL: 'bool';
TYPE_BYTE: 'byte';
TYPE_CHAR: 'char';
TYPE_INT16: 'int16';
TYPE_UINT16: 'uint16';
TYPE_INT32: 'int32';
TYPE_UINT32: 'uint32';
TYPE_INT64: 'int64';
TYPE_UINT64: 'uint64';
TYPE_BINARY: 'binary';
TYPE_FLOAT32: 'float32';
TYPE_FLOAT64: 'float64';
TYPE_DOUBLE: 'double';
TYPE_STRING: 'string';

LITERAL
: (('"' ~'"'* '"') | ('\'' ~'\''* '\''))
;

IDENTIFIER
: (LETTER | '_') (LETTER | DIGIT | '.' | '_')*
;

COMMA
: ','
;

fragment LETTER
: 'A'..'Z' | 'a'..'z'
;

fragment DIGIT
: '0'..'9'
;
```

Listing B.22. Antlr grammar for dynADL(continued)

```
fragment HEX_DIGIT
: DIGIT | 'A'..'F' | 'a'..'f'
;

WS
: (' ' | '\t' | '\r' '\n' | '\n')+ -> channel(HIDDEN)
;

SL_COMMENT
: ('//' | '#') (~'\n')* ('\r')? '\n' -> channel(HIDDEN)
;

ML_COMMENT
: '/*' .*? '*/' -> channel(HIDDEN)
;
```

VITA

Background

Anton D. Hristozov holds an Electrical Engineering degree from the Technical University of Sofia, Bulgaria. He later graduated from the University of Pittsburgh with a Masters in Telecommunications and Information Science. He is finalizing his doctoral degree in Technology at Purdue University. He works as a research engineer at the Software Engineering Institute Carnegie Mellon University. He is involved with scientific research in the field of safety assurance of real-time systems. His research interests involve embedded and real time systems, including robotic systems. He is a Linux user and enjoys working with cyber physical systems which use sensors and physical phenomena. He has worked on various types of UAVs and UGVs with focus on mission critical and fault-tolerant software. Anton is generally interested in runtime software assurance and better security, reliability and flexibility of cyber physical systems.

Professinal Achievements

Created an Architecture Description Language for Dynamic Management of Software Components

- Developed Antlr grammar and parser
- Created a code generation engine that outputs efficient C++ code for component management and initialization
- Provide support for Posix and Ros platforms

Worked on enhancements of the PX4 autopilot software architecture

- State recovery for running software components
- Redundant components
- Resistance against software attacks

- Dynamic reconfiguration during run-time

Developed a secure framework for a UGV, based on Raspberry Pi

- Woked on implementation of hypervisor extensions for sensor driver
- Developed software attack scenarios to attack the line tracking sensor
- Implemented attack proof solutions for the UGV

Built a drone from scratch

- Researched all electrical and mechanical parts
- Worked with PX4 autopilot
- Conducted experiments in lab environment

Implemented a software rejuvenation for an embedded system

- Worked in VxWorks environment
- Modified the VxWorks kernel to support checkpoints and restore mechanisms for user space processes
- Integrated the solution in a complete system

Taught Engineering Courses at CCAC, Pittsburgh and Technical University, Sofia as adjunct faculty

- C programming
- Computer Architecture
- Analysis and Design of Software Systems
- Practical Control Solutions

Developed a Digital Signal Processing (DSP) Simulator

- Researched algorithms
- Implemented algorithms and graphical tools to display the results from the simulation
- Used Pascal and C++ for the development

Skills

Technical Skills

- Real Time Operating Systems (VxWorks,Integrity,OSE,QNX)
- Linux development, user space and kernel space
- C and C++ programming
- Python and shell scripting
- Embedded systems, including robotic systems, drones and other vehicles
- PX4 autopilot architecture
- Hypervisor architecture and development
- Hardware knowledge of electronic devices and embedded systems
- Writing technical documentation
- Writing scientific papers and performing research

Leadership Skills

- Project management
- Interviewing technical candidates
- Mentoring engineers
- Working in a distributed environment, including outsourcing

Work History

- Research Engineer Software Engineering Institute Carnegie Mellon University, Pittsburgh, PA Oct. 2018 until present
- Firmware Engineer NetApp Inc. Pittsburgh, PA Apr. 2011 Oct. 2018

- Senior Software Engineer Ansaldo STS Pittsburgh, PA Jan 2005 Apr. 2011
- Senior Software Engineer Compunetix Pittsburgh, PA Feb 2001 Jan 2005
- Team Lead TouchTown (startup) Pittsburgh, PA Mar. 2000 Jan 2001
- Quality Assurance Claritech (startup) Pittsburgh, PA Oct 1998 Mar. 2000
- Team lead Intransco (startup) Sofia, Bulgaria and Austin,Tx Oct 1996 Oct 1998
- Research Associate Bulgarian Academy of Science Sofia, Bulgaria Oct 1989 - Oct 1996

Education

- Bachelors of Science in Electronics and Automation Technical University Sofia, Bulgaria June 1989
- Masters of Telecommunications and Information Science University of Pittsburgh, PA 2007
- Doctor of Technology Candidate Purdue University, Indiana Expected graduation Spring 2023

Languages

- English (fluent)
- Bulgarian (native)
- Russian (fluent)
- French (working knowledge)

Publications

- Practical, Provable, End-to-End Guarantees on Commodity Heterogeneous Interconnected Computing (CHIC) PlatformsPractical, Provable, End-to-End Guarantees on

Commodity Heterogeneous Interconnected Computing (CHIC) Platforms, HotSOS Hot Topics of Science of Security, Apr 16, 2021

- TwinOps - DevOps meets model-based engineering and digital twins for the engineering of CPSTwinOps - DevOps meets model-based engineering and digital twins for the engineering of CPS, Models 20: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering languages and Systems, Oct 1, 2020
- The Software Communications Architecture. "Resource management in Future Internet,"The Software Communications Architecture. "Resource management in Future Internet," River Publishers, Mar 27, 2013
- Sensor Data Protection in Cyber-Physical Systems, 17th Conference on Computer Science and Intelligence Systems, Sept. 2022
- Component Interface Standardization in Robotic Systems, 17th Conference on Computer Science and Intelligence Systems, Sept. 2022
- A Methodology for Estimation of Software Architectural Complexity in Publish-Subscribe Systems, International Conference Automatics and Informatics 2022, Oct. 2022
- Resilient Architecture Framework for Robotic Systems, International Conference Automatics and Informatics 2022, Oct. 2022
- Reviewing the role of machine learning and artificial intelligence for remote attestation in 5G+ networks, 5G and Future Networks Technologies, Oct. 2022
- Security of Cyber-Physical Systems through Dynamic Component Management, IAJC Conference 2022, Ocotber 2022
- Secure Robotic Vehicles: Vulnerabilities and Mitigation Strategies, 2022 Virtual IEEE International Symposium on Technologies for Homeland Security, November 2022
- Software Rejuvenation for Safe Operation of Cyber-Physical Systems in the Presence of Run-time Cyber Attacks, IEEE Transactions on Control Systems Technology, Jan. 2023

INDEX