Communication Papers of the of the 17th Conference on Computer Science and Intelligence Systems pp. 305–312
DOI: 10.15439/2022F7
ISSN 2300-5963 ACSIS, Vol. 32

Component Interface Standardization in Robotic Systems

1st Anton Hristozov
Polytechnic Institute
Purdue University
West Lafayette, Indiana, USA
ahristoz@purdue.edu

2nd Dr. Eric Matson
Polytechnic Institute
Purdue University
West Lafayette, Indiana, USA
ematson@purdue.edu

3rd Dr. Eric Dietz
Polytechnic Institute
Purdue University
West Lafayette, Indiana, USA
jedietz@purdue.edu

4th Dr. Marcus Rogers
Polytechnic Institute
Purdue University
West Lafayette, Indiana, USA
rogersmk@purdue.edu

***Abstract*—Components are heavily used in software systems, including robotic systems. The growth in sophistication and diversity of new capabilities for robotic systems presents new challenges to their architectures. Their complexity is growing exponentially with the advent of AI, smart sensors, and the complex tasks they have to accomplish. Such complexity requires a more flexible approach to creating, using, and interoperability of software components. The issue is exacerbated because robotic systems increasingly rely on third-party components for specific functions. In order to achieve this kind of interoperability, including dynamic component replacement, we need a way to standardize their interfaces. We desperately need a formal approach for specifying what an interface of a robotic software component should contain. This study analyzes the issue and presents a universal and generic approach to standardizing component interfaces for robotic systems. Our approach is inspired and influenced by well-established robotic architectures such as ROS, PX4, and Ardupilot. The study also applies to other software systems with similar characteristics to robotic systems. We consider using JSON or Domain-Specific Languages (DSL) development with tools such as Antlr or Xtext and automatic code and configuration files generation for frameworks such as ROS and PX4. A case study with ROS2 has been done as a proof of concept for the proposed methodology.**

***Index Terms*—CPS, robots, software architecture, interface, ROS, autopilot**

I. Introduction

COMPONENT interfaces have functional and non-functional parts [1]. Functional characteristics are the ones that usually receive the most attention and are considered most important. These characteristics cannot be completely independent of the overall architecture [2], and in this work, we concentrate on systems that use popular architectural paradigms such as ROS [3]. Having interfaces in explicit form is very important for analyzing and achieving component replacement and maintenance during run-time for maintenance or other objectives such as adaptation. They are important during development, too, when software components are given to third parties or bought from other organizations. Many systems do not have their interfaces presented in an explicit form in one central place. Their interfaces are dispersed in text configuration files and source code and are implicit rather than explicit, making their understanding and analysis more difficult. This is the case for systems we studied, such as ROS, PX4, and Ardupilot, as typical representative systems from the robotic field.

A. The need for standardization of robotic interfaces

Robotic systems have had typically different forms of custom implementations until recently. This has been steadily changing in the last decade, and robotic architectures are becoming more consistent and based on existing frameworks. Even when they are based on architectures such as Autosar [4] though they still do not offer a consistent model for providing interfaces for components that can become interoperable within the architecture or even between different architectures. This creates a need for a universal and consistent proposal for interface specification that can be complete and expressive and bring us closer to better interoperability among vendors of software components. This can also help us reach the possibility of better component reuse and system evolution.

In this paper, a general representation of component interfaces is proposed. The main contribution is defining a universal interface specification for software components applicable to components used in robotic systems. Another contribution is using a DSL to represent such interfaces and compare them and generate code and configuration files for different architectures. The applicability of the idea is demonstrated through a case study in ROS.

The following two sections talk about possible structural representations of interfaces through JSON and a DSL. The component interface standardization section proposes a universal interface representation and its constituent parts. The dynamic component management section defines the conditions for component interfaces to be compatible. An example is shown based on an extension of the Thrift IDL grammar. Furthermore, a case study with ROS illustrates the approach for a modern robotics framework.

B. Publish-subscribe robotic systems

Today, many robotic systems use an asynchronous architecture for message delivery based on the publish-subscribe paradigm. An example is PX4 [5] and ROS [6]. These systems are easier to use and provide many benefits for quickly adding new interfaces and components through a loosely coupled architecture. We believe that the interface standardization we

are presenting in this work will be used in a system that uses either a publish-subscribe broker or a similar message delivery mechanism. This decision is also based on the fact that publish-subscribe is common in other autopilot software systems, for example, Ardupilot. The central capability in this paradigm is that we can have software components publish topics or subscribe to topics making an arbitrary graph of connections between the components in the architecture. In this respect, we have a many-to-many relationship between components which can also be supported dynamically by components subscribing to messages at different times or advertising messages that they will publish later. This enables the possibility of adding new components at run-time and removing obsolete components, given that we follow specific rules.

C. ROS interfaces

We will focus our ROS investigations on ROS2, which is the latest version of the framework. ROS2 has been improved compared to ROS1 and is holding new promise for the future of robotics [3]. The improvements in how the system is configured and its real-time characteristics will allow new classes of applications to be possible. In addition to messages and services, ROS2 includes the concept of actions that are particularly applicable to robots in addition to messages and services. Therefore the three most essential features that ROS supports are messages, services, and actions. They are strongly related to how an interface can be described and used in our interface formulation in the next section. These interfaces can be used to achieve dynamic component exchange during run-time as part of the pursuit of fault-tolerant systems and systems that can adapt to changing conditions [7].

II. Component Interface Standardization

It is necessary to create a framework of components that can be interchangeable and compatible in a particular architecture or even between architectures. We aim to derive a comprehensive interface standardization that describes the component's interfaces. This standardization does not deal with the timing and memory characteristics of the component, which is essential but will not be addressed in this work.

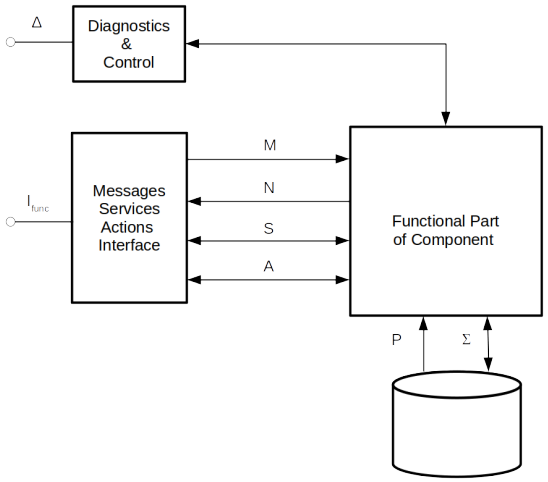

Fig. 1. Interface Standardization Diagram

Figure 1 depicts how a typical component's interface in a robotic system should look. The diagnostics portion is shown as a separate box as it can be a separate thread of execution and possess an interface Δ dedicated to diagnostics. The configuration parameters P and the component's state Σ use nonvolatile storage. The incoming messages M and outgoing N and service S and actions A are how the component communicates with other components in the system.

An interface I is a heterogeneous set formed by the union of several sets. The set I describes all common characteristics for robotic systems and even other software systems' components. Our generalization of an interface for any component is defined by equation 1.

$$I = M \cup N \cup S \cup A \cup P \cup \Sigma \cup \Delta \tag{1}$$

, where I is the interface of a component,
M is the set of subscribed messages
N is the set of published messages
S is the set of services
A is the set of actions
P is the set of configuration parameters
Σ is the set of state variables
Δ is the set of diagnostics services

A. Messages

The publish-subscribe paradigm allows many components to subscribe to any published messages. We consider two main types of messages in our analysis in a publish-subscribe model [8] or in a general case of a system using messaging. In our equation, the first set denoted as M is the set of messages the component subscribes to. The other set of messages is the messages N that the component publishes for other components to receive. We assume that the relationship is many to many with regards to both sets, and any component can subscribe to any messages and publish any messages that any other component can receive [5]. The dispatching of messages is typically handled by a message broker component that

performs the necessary bookkeeping and queuing to ensure delivery.

B. Services

Services are based on client-server interactions of one component making a service call and another component servicing this call. The client component blocks until the service finishes executing. Then the result is returned. Services can be characterized by parameters sent and results returned to the caller. The timing of services will not be part of the interface specification in this work. We concentrate on the parameters and results and assume that the service always completes with a result returned to the caller in a finite time.

C. Actions

Actions are particularly useful for some robotic operations and are supported in ROS2 [9]. This includes operations that are started, monitored, and getting a notification when the goal is reached. This allows the requester of the action to do something else while the action continues while receiving notifications asynchronously from the executor of the action. They are using messages and services to accomplish the action and thus provide a more complicated interaction and are suitable for tasks that need to continually send feedback to the client of the action. They have a goal defined as a service, feedback messages periodically sent from the action server to the client, and a result defined as another service. This ROS2 model is general enough for defining actions that can be used in many practical scenarios.

D. Configuration Parameters

The set of configuration parameters is used to configure a component at startup during the initialization process. They provide initial values for variables and constants used in the component during its lifetime. Typically these values come from a configuration file or are embedded in the source code. The parameters determine the initial state of the component and its operation. That is why they need to be part of the full interface description. A universal approach to configuration parameters is to keep them out of the source code in the form of a configuration file, preferably in a structured fashion. A possible structured format can be XML, JSON, or even some custom configuration language format.

E. State

Almost every component has some state that continually changes as the component operates as part of the system. Describing the component's state allows us to build tools that can save and recover it when needed. This is especially important when new components replace old ones while the system works. The state needs to be captured in a language-neutral form since different components may have different implementations using different technologies and languages. State variables with complex data types can be saved and retrieved using serialization and deserialization techniques. This makes it possible to restore precisely the state of every variable in a component, independently of how complex its type is. The description of a component's state at a particular point makes it transparent and can lead to new possibilities where a new component can pick up work that an old component has started without interrupting the flow of operations in a complex system. This can be a considerable advantage in architectures where the system cannot be halted and needs to be incrementally upgraded.

F. Diagnostics and Control

For a component to be useful in software architecture, it is good to have a self-diagnostics section with a well-defined interface. It makes sense to have the diagnostics work in an independent thread of execution so that it does not interfere with the main execution thread of the component. This way, other components can send queries and commands to a running component without affecting its main thread of execution. Having a separate thread for diagnostics makes it practical to have diagnostics services that can block in the requesting client and wait for a response from the diagnostics thread. This part of the interface can also be used to control a component's state, for example, running, ready, and others. Having this ability can allow the user to stop a component, save its state, load another component in its place and switch them. This adds additional non-functional abilities that can lead to many exciting scenarios.

The diagnostics and control interface should allow for control of the internal state machine of a component. If a component is in the middle of an operation that cannot be interrupted, a request to this interface can request a pause of the activities when the current operation is done. This way, we can have the notion of work transactions that cannot be interrupted and can have the opportunity to stop the component, save its state and then can unload it and load a replacement component using the saved state to continue. The approach of controlling the state machine of a component provides flexibility since we want to allow for operations to reach a steady state and results to be published before we perform any dynamic reconfiguration action in the system [10] [11].

G. Contracts in Component Interfaces

The software contract paradigm has been popular for a long time. In most cases, though, the focus has been to look at contracts in the scope of classes and functions [12]. This is appropriate for object-oriented systems and relations inside a component but is not universal for components that use different languages and are integrated based on their interfaces. There is a way to relate the software contract approach with component interfaces [13]. Our approach to embedding assumptions and guarantees into the interface makes the notion of contracts possible for software components independently of their implementation.

$$C = (A, G) \tag{2}$$

The general representation of contracts can be given with equation 2 where A represents the assumptions and G the guarantees [14]. We can apply this definition to our notion of contracts and their relation to the general interface representation we propose. The sets of services S and actions A have assumptions and guarantees since they send requests and expect responses. The messages M we subscribe to can be assumptions, and the messages N we publish are the guarantees that we promise to fulfill. The diagnostics interface also has assumptions in the interface Δ with the services it handles, and the guarantees are the responses to the user of this interface. Similarly, the configuration parameters P are only assumptions, and the state Σ has both assumptions and guarantees since we are saving and restoring it from external storage to the component. This mapping of contract obligations into the interfaces allows us to use the design by contract paradigm at the software component level.

III. Dynamic Component Management

Creating dynamically reconfigurable systems is challenging mainly because the interfaces of components are not explicitly defined. Most systems have a static architecture that does not allow for dynamic reconfiguration [15] without modifications. Systems based on publish-subscribe architectures can be extended, although many challenges need to be overcome. One of the really compelling challenges is to transfer the application state into the new instances of components from existing components [16]. This necessitates a well-defined interface that includes the description of the state. Dealing with state storage and retrieval assumes a mechanism that performs serialization and deserialization of state variables so that they are stored in nonvolatile memory.

Another challenge in dynamic reconfiguration is the impact of component replacement on the entire architecture. This effect can differ based on the component and its place in the architecture. One way to model this dependency is to create a graph where the nodes are the components, and the edges are the connections between them [17] [11]. Maintaining such a dynamic graph shows the components that can be affected by changes in a particular component. In the case of publish-subscribe, components are only coupled through messages, and this makes it easier to perform dynamic changes. This architecture of loosely coupled parts makes it easy to add and remove components at run-time. Messages are buffered by the message broker and are delivered to the component only when it is ready to retrieve them. We can also control the internal state of any component by using the diagnostics and control interface.

Another issue we have to consider for the management of components is the security of dynamically loaded components since they need to be trusted before being used [4]. The interface alone cannot solve the security, and other techniques need to be utilized. Adding some form of authentication may be needed to make sure the component can be trusted before it is loaded and run in the system. A dedicated component manager module is proposed in several works [18]. Some works propose a dedicated language for specifying component capabilities [19].

A. Interfaces and Their Role in Component Management

It is important to analyze when a new component can be a candidate for the replacement of an existing running component. Having an explicit and comprehensive interface is very important to accomplish this. We use set theory to formalize the interface definition and its constituent parts. If a component is a superset of another component, it can potentially replace it as shown in equation 3 where I_2 is the new component, and I_1 is the old component. Equation 3 is a necessary but not sufficient condition for a successful replacement.

$$\begin{aligned} I_1 &\subset I_2 \\ &or \\ M_1 &\subset M_2 \\ N_1 &\subset N_2 \\ S_1 &\subset S_2 \\ A_1 &\subset A_2 \\ P_1 &\subset P_2 \\ \Sigma_1 &\subset \Sigma_2 \\ \Delta_1 &\subset \Delta_2 \end{aligned} \tag{3}$$

A more precise definition of when a component is a successful candidate to replace another component is shown by equation 4. We show that all sets can be subsets of the new component sets, but only if the extra elements in the new sets are not used. Equation 4 can be further restricted to the equivalency case shown by equation 5 where the two components are equivalent. Equation 6 shows the conditions for equation 5 to hold true. We can consider that 6 provides a necessary and sufficient condition to achieve component compatibility. This means that all of the equivalence relationships in equation 6 should be true so that we can satisfy equation 5. The equivalence of each set in equation 5 means that they have identical number and type of elements.

$$\begin{aligned} M_1 \subset M_2, \text{ given } M_2 - M_1 \text{ is not used} \\ N_1 \subset N_2, \text{ given } N_2 - N_1 \text{ is not used} \\ S_1 \subset S_2, \text{ given } S_2 - S_1 \text{ is not used} \\ A_1 \subset A_2, \text{ given } A_2 - A_1 \text{ is not used} \\ P_1 \subset P_2, \text{ given } P_2 - P_1 \text{ is not used} \\ \Sigma_1 \subset \Sigma_2, \text{ given } \Sigma_2 - \Sigma_1 \text{ is not used} \\ \Delta_1 \subset \Delta_2, \text{ given } \Delta_2 - \Delta_1 \text{ is not used} \end{aligned} \tag{4}$$

$$I_1 \equiv I_2 \tag{5}$$

$$\begin{aligned} M_1 &\equiv M_2 \\ N_1 &\equiv N_2 \\ S_1 &\equiv S_2 \\ A_1 &\equiv A_2 \\ P_1 &\equiv P_2 \\ \Sigma_1 &\equiv \Sigma_2 \\ \Delta_1 &\equiv \Delta_2 \end{aligned} \tag{6}$$

IV. Interface Representation using Json

One of the goals of capturing an interface formally is to be able to process it programmatically and make decisions at run time. Another goal is to have the interface in one place, for example, in a textual interface file. This is a significant step compared to having interfaces defined in source code and in multiple text files with different formats. There are many choices in selecting an approach for interface notation. One of them is to use formats like JSON, which is easy to parse and flexible enough for this task. A more ambitious possibility is to design a unique language to represent interfaces and have a parser that can analyze it automatically. For this study, we will use both approaches but will start with the JSON approach as our goal is to illustrate the content, not so much the format of the interface representation.

A JSON representation in a dedicated file can include all parts of an interface I in one place and keep the information in a structured form. JSON allows for the definition of complex types based on the common types, and this can be used when defining interfaces for real systems. One can use key-value pairs that define each data element's name and data type. Parsing such an interface file in JSON format becomes practical, and comparing components for their equivalence with regard to their interfaces becomes possible. One potential drawback is that the representation in this format can be rather verbose and tedious to create, but for the parsing, it really does not make a difference.

V. Domain Specific Languages for Interface Representation

Using a domain-specific language to represent an interface specification has many benefits. The main benefit is that the interface can be created using a textual language that has well-defined grammar. This textual representation has some benefits over a graphical language such as UML as it improves the formalization of the interface creation and forces users to comply with grammar rules and the associated tools that enforce them. A second benefit is that the parser can generate code and configuration files automatically while parsing the interface file. This can help automate the process of the software component creation, provide a more unified and automated approach, and reduce the margin of error, especially when different vendors do component development. Designing our DSL can be tailored to the needs of the systems. Having a language grammar imposes stricter interface specifications which makes any automation easier. The requirements we have defined for such a DSL are the following:

- to be able to represent all aspects of the component interface fully as shown in figure 1
- to be easy to read by humans
- to be in a structured format so that it can be parsed easily by automatic tools
- to be expandable in order to support future needs

A. Development Approach

Domain-specific languages are becoming easier to create by using special tools that can create parsers. One such tool is Antlr [20] although one can use any other tool that they are familiar with. The generation of parsers that are created from an Antlr grammar file is possible in several languages such as Java and C++. The grammar files used by Antlr have a traditional BNF notation. The result is a working C++ or Java parser that can be used to parse a file that complies with the grammar. Adding new features to the language happens by modifying the grammar and regeneration of C++/Java files. Many existing grammars can be downloaded and adapted as per the user's needs. Alternatively, Xtext can be used as a newer tool for creating DSLs [21]. It provides a better set of tools for the task.

B. Code Generation

Components in different frameworks are defined differently for reasons such as what architecture and programming languages are used. Having an interface file in the form of a DSL makes it easier to add code and configuration files automatically. For example, in the case of ROS, all services are created as .srv files in a folder with the name srv. Similarly, actions are in a folder with a file for each action with an .action extension. Messages are in a msg folder with a file for each message with .msg extension. All these files can be easily generated from a single interface file.

The concept of code generation can be extended to more than one platform. For example, PX4 also has msg files but does not support srv and actions, and the declarations for message topics follow a standard pattern that exists in the C++ code. This boilerplate code can be easily generated from the interface representation in a DSL. The parsing and validation of the syntax of the interface is an independent step that a target-specific code generation can follow that users can add for their specific platforms. This approach looks practical for systems with well-defined architectural patterns such as ROS, PX4, and Ardupilot.

C. Example DSL Grammar

Developing a proprietary DSL may be a time-consuming task. A practical approach is to start with an existing Antlr grammar and enhance it if needed. There are different possibilities to use. One option is to use the Apache Thrift grammar as a well-established interface definition language(IDL) [22]. Thrift already has the options to create structures and services sufficient for what is needed when creating an interface. An

equivalent interface to our earlier example is shown using the enhanced syntax based on Apache Thrift grammar, giving us a new DSL grammar in Listing 1. Using Thrift-like language

Listing 1 DSL Example Component Interface File

```
message message1 {
   string str,
   i16 i
}
message message2 {
   string str,
   float f
}
service service1 {
   string str,
   i32 i,
   string result
}
service service2 {
   i32 i,
   float f
}
action action1 {
   i32 par1,
   string s,
   string r
}
action action2 {
   double par1,
   float par2,
   string f,
   i16 result
}
configuration params{
   string param1,
   i32 param2,
   float param3,
   double param4
}
state state_variables {
   float position,
   float velocity,
   float acceleration,
   string mode
}
service diagnostics{
   string result,
   i32 r,
   string cmd
}
```

is even more intuitive, more compact, and provides the benefit of starting with a grammar that is available as an open-source file. The example above shows that Thrift may be sufficient in most cases, although its grammar can be easily extended, and a new C++ or Java parser can be developed if needed by using Antlr. The best part of having a parser is that we can add code generation based on our interface file and thus improve and standardize the development process.

The original grammar had to be enhanced with the following sections to allow us to support: actions, messages, configuration, and state portions of the interface specification, as shown in Listing 2. This allows us to parse an interface file with all the earlier sections. The file parsing confirms that the interface conforms to the rules of the defined language. In addition, we can easily generate configuration files specific to a particular platform, for example, ROS.

Listing 2 Grammar Additions to Thrift

```
definition
: const_rule | typedef_ | enum_rule |
senum | struct_ | union_ |
exception_cpp | service | action |
configuration | message | state;

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

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

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

state
: 'state' IDENTIFIER
'{' field* '}' type_annotations?;
```

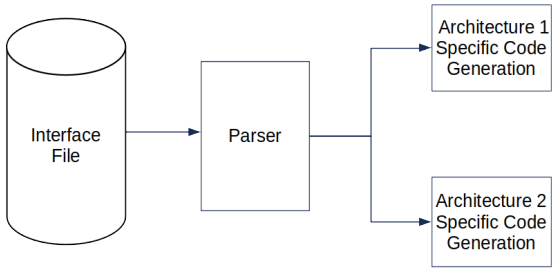

Fig. 2. Code Generation from Interface File

The code generation is shown in figure 2. The parser uses a listener or visitor pattern, allowing it to intercept each token

and make decisions based on what is necessary for the parsing logic. Since parsing is based on the grammar shown above, this is independent of the software architecture. On the other hand, the code generation portion depends on whether we need to create configuration files for ROS or for PX4, for example. The code generation architecture blocks can be added in the future depending on what new architectures need to be supported.

The most significant advantage of the formal interface design is that the interface allows for documentation of the component and for its comparison with other components with possible reuse and integration within an existing system. In addition to configuration files, we can generate code that reflects the interface design. Some systems have their interfaces in code instead of external configuration files, for example, PX4. Code generation is also applicable for the diagnostics section, where we can have a template code for diagnostics services. Even with boilerplate code generation, we will always need some manual code development.

D. Case Study with ROS

Our case study focused on how a ROS component can be represented consistently through an interface file. The prototype we developed aimed to parse the interface file and determine if it complies with the grammar of the IDL used, in this case, the Thrift-like derivative DSL we propose. Secondary, the parsing process generated configuration files for a ROS process automatically. These included message files, service files, and action files. We did not generate any source code since this is not part of the interface. The study showed that a parser could be used successfully for custom interface definition languages, and the sections of our interface standardization worked well in this experiment.

The advantages of representing the interfaces in the proposed way are that instead of using multiple files for different aspects of the interface and having some of them in the source code, we have moved the specification to a single textual file that can be parsed automatically by tools. This allows us to easily compare components from different vendors and to reason if they are compatible. Such analysis otherwise is pretty difficult and cannot be done automatically. The problem is exacerbated when components from different architectures are to be compared, where architectural specifics make the task even more difficult. The same methodology can be applied to different architectures, provided they have the same functional elements such as messages, services, and actions. The cost in time for parsing a component interface before it is loaded is not an issue. In fact, ROS does that in a more complicated manner because it has to parse multiple files and connect the data with the source code.

A promising future direction is to develop an architecture-specific tool that allows for a defined interface that can generate all the boilerplate code specific to that architecture. We described a possible implementation for ROS, but similarly, the code generation can be tailored to other systems that use different ways to represent their interfaces. This can help design components that need to be ported from one system to another. Reusing third-party components will only continue to increase, and ways to minimize efforts and introduce a common language can only help design new and expanding existing architectures.

VI. Conclusion

Interfaces represent components and their capabilities. Universally presenting them can make reuse possible and dynamic component reconfiguration at run-time and design-time. A viable first step in this direction is to formalize the representation of interfaces to make this representation easier to parse programmatically and to use as a basis for run-time architectural decisions. The presented standardization can be used in different architectures as it is pretty general and covers many component characteristics. It directly applies to systems that use ROS/ROS2 or even autopilots such as PX4 and Ardupilot. Our generic approach to defining interfaces for robotic systems can be used universally and help achieve better and more resilient architectures with new capabilities for reconfiguration and maintenance.

References

[1] 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.

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

[3] I. Malavolta, G. Lewis, B. Schmerl, P. Lago, and D. Garlan, "How do you architect your robots? state of the practice and guidelines for ros-based systems," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 31–40.

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

[5] 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.

[6] M. Lauer, M. Amy, J.-C. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Engineering adaptive fault-tolerance mechanisms for resilient computing on ros," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, 2016, pp. 94–101.

[7] 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.

[8] 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.

[9] E. Erős, M. Dahl, K. Bengtsson, A. Hanna, and P. Falkman, "A ros2 based communication architecture for control in collaborative and intelligent automation systems," *Procedia Manufacturing*, vol. 38, pp. 349–357, 2019, 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM 2019), June 24-28, 2019, Limerick, Ireland, Beyond Industry 4.0: Industrial Advances, Engineering Education and Intelligent Manufacturing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2351978920300469

[10] N. T. Huynh, "An analysis view of component-based software architecture reconfiguration," 03 2019, pp. 1–6.

[11] A. Butting, R. Heim, O. Kautz, J. O. Ringert, B. Rumpe, and A. Wortmann, "A classification of dynamic reconfiguration in component and connector architecture description," in *MODELS*, 2017.

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

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

[14] 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.

[15] D. de Leng and F. Heintz, "Dyknow: A dynamically reconfigurable stream reasoning framework as an extension to the robot operating system," in *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2016, pp. 55–60.

[16] C. Cu, R. Culver, and Y. Zheng, "Dynamic architecture-implementation mapping for architecture-based runtime software adaptation." in *SEKE*, 2020, pp. 135–140.

[17] 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.

[18] S. Alhazbi and A. B. Jantan, "Safe runtime reconfiguration in component-based software systems," in *Software Engineering Research and Practice*, 2008.

[19] T. V. Batista, A. Joolia, and G. Coulson, "Managing dynamic reconfiguration in component-based systems," in *EWSA*, 2005.

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

[21] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 307–309. [Online]. Available: https://doi.org/10.1145/1869542.1869625

[22] K. Grochowski, M. Breiter, and R. Nowak, "Serialization in object-oriented programming languages," in *Introduction to Data Science and Machine Learning*, K. Sud, P. Erdogmus, and S. Kadry, Eds. Rijeka: IntechOpen, 2020, ch. 12. [Online]. Available: https://doi.org/10.5772/intechopen.86917