Coping with Architectural Mismatch in Autonomous Mobile Robotics

Carle Côté, Roger Champagne and François Michaud

Abstract—Integration of independently developed software components is common nowadays in autonomous mobile robotics. However, the field faces challenges similar to those faced by the Software Engineering community, namely that assembling software components of various sources to build a larger system has limited success. The term architectural mismatch has been used to explain this phenomenon. This paper studies the concept of architectural mismatch in the context of software development for autonomous mobile robotics, and describes how a programming framework designed specifically for this field, MARIE, addresses some of the related issues. Our experience with MARIE in the design of a socially interactive autonomous mobile robot allows us to conclude that this framework provides interesting solutions to the problem of mismatch repairing.

I. INTRODUCTION

Software integration is one (if not the most) critical aspect in designing software for autonomous mobile robotic systems, as shown in the recent publications on the topic [1], [2]. Component-based development (CBD) is one solution in which functionalities are encapsulated in reusable software elements called components. Components are then configured and interconnected to implement the desired system, with the hope of improving architectural issues related to software quality attributes such as reusability, maintainability, extensibility, and flexibility, and business quality attributes such as time-to-market rollout schedule and integration with legacy systems [3].

An important shortcoming of CBD is that building large systems using reusable components from various sources is not a straightforward task. A decade ago, Garlan et al. [4] coined the term architectural mismatch to explain why the software community has had limited success in building software systems from reusable components, and CBD is still struggling pretty much with the same challenges today. These challenges are also shared by the robotics community, which could benefit from the solutions identified so far by the software engineering community.

Therefore, this paper presents architectural challenges and design solutions to cope with architectural mismatch in component-based software architectures for autonomous mobile robots. The paper is organized as follows. Section II presents the architectural mismatch concept and its impacts on robotic software development. Section III explains how MARIE (Mobile and Autonomous Robotics Integration Environment), our distributed component-based software architecture, implements design solutions addressing issues related to repairing architectural mismatch. Section IV explains how architectural mismatch is handled in real case studies encountered in the development of a socially interactive autonomous mobile robot operating in real life settings. The paper concludes with remarks about how robotic software design can benefit from identifying architectural mismatch early in the development process, and how an approach such as the one used in MARIE can contribute to a better understanding of the underlying requirements of robotic software design challenges.

II. COMPONENT-BASED SOFTWARE DEVELOPMENT AND ARCHITECTURAL MISMATCH

A. Software Engineering Concepts

In an attempt to improve reusability, the Software Engineering community has been trying to exploit a very basic design principle: build systems (in this case, software) by assembling existing parts, instead of starting from scratch every time. This approach is typically termed “Component-Based Development” (CBD) or “Component-Based Software Engineering” (CBSE) and is supported by a large research community [5], [6]. Various industrial standards and frameworks, such as CORBA [7], COM [8] and J2EE [9] (recently renamed JEE), have been specifically designed for building software applications from assembly of pre-existing reusable components. Although the basic idea is simple, its actual implementation has had limited success. Garlan et al. proposed an explanation for this limited success in a seminal paper [4], in which they introduced the notion of architectural mismatch:

“Architectural mismatch stems from mismatched assumptions a reusable part makes about the structure of the system it is to be a part of. These assumptions often conflict with the assumptions of other parts and are almost always implicit, making them extremely difficult to analyze before building the system.”

Using a case study consisting in assembling a few components to build a software system, Garlan et al. summarized the assumptions which can contribute to architectural mismatch by defining four categories: 1) the nature of the components; 2) the nature of the connectors; 3) the global architectural structure; and 4) the construction process. The authors also proposed four topics for future research which might lead to resolving architectural mismatch: 1) make
architectural assumptions explicit; 2) construct large pieces of software using orthogonal subcomponents; 3) provide techniques for bridging mismatches; and 4) develop sources of architectural design guidance.

More recently, Bass et al. [3] stated that architectural mismatch is actually a specific case of interface mismatch. Here, the term interface refers to all the assumptions components can make about each other. This is a much broader definition than what is typically considered as a software component’s interface, i.e., its Application Programming Interface (API). Bass et al. also present three broad categories of techniques required to deal with interface mismatch, which are:

1) techniques to detect mismatches (one needs to know that there is a problem before fixing it); 2) techniques to repair mismatches; and 3) techniques to avoid mismatches.

The obvious solution to fix mismatched components is to alter them until they fit. This solution can however prove to be suboptimal, as changing existing components may take more time than rewriting them from scratch. Also, one of the underlying assumptions about software components is that they are built by experts in the given field. Altering a component may break some of the underlying design assumptions (which are often not visible), changing its function or leading to unexpected bugs. This type of fix also assumes that the source code of the component is available for modifications, which is not always the case. Instead, given that mismatches exist and that modifying existing components may not always be desirable or even feasible, Bass et al. propose three techniques to repair them:

1) Wrappers: A wrapper typically encapsulates an existing component and modifies its interface. The original component is not altered in any way; it is simply encapsulated inside a new component and its interface is accessed via the interface of the new component.

2) Bridges: A bridge basically creates a new component which is inserted in between two (or more) ‘incompatible’ components. Bass et al. define a bridge as follows: "A bridge translates some required assumptions of one arbitrary component to provide assumptions of another". For example, if a component produces its output in Postscript format and this information is required by a component that can only read the Portable Document Format (PDF), then one solution consists in implementing a bridge which will convert Postscript to PDF. The bridge is not encapsulated in any of the components it connects, and it must be instantiated by an outside agent (perhaps one of the components it connects, but not necessarily).

3) Mediators: A mediator is an abstraction that exhibits the characteristics of both wrappers and bridges. Its most distinguishing characteristics resides in its planning capability. Wrappers and bridges must know at compile time which components they will be involved with. A mediator, however, can dynamically determine at runtime which wrappers and/or bridges are required, and can assemble them to other components based on such context.

B. Architectural Mismatch in Autonomous Mobile Robotics

When considering the current development context in robotics, the presence of architectural mismatches is almost inevitable. We have identified three important sources of mismatched assumptions.

The first source appears with the integration of components coming from various sources. Designing highly sophisticated autonomous mobile robots requires the integration of capabilities usually developed independently, such as localization and mapping, navigation, visual tracking, speech recognition, signal processing, planning, and human-robot interface, just to name a few [10], [11]. The appropriate use of heterogeneous software applications is important for efficient scientific and incremental progress of the field, to avoid reinventing what is made available by others and focusing efforts towards making new discoveries [10], [12]. However, these heterogeneous software applications usually address different design and implementation requirements, which are most of the time not explicitly documented. This complicates the integration work. Integration issues based on architectural assumptions can be easily observed just by looking at the wide variety of communication protocols and mechanisms, operating systems, robotics platforms, programming languages, and design principles used to create these applications.

The second source comes from the absence of broadly adopted component development standards, methodologies or frameworks by the robotics community. Component integration is more like an art (rather than being a formalized process), heavily based on trial and error and on the developer’s own experience on how to handle underlying integration issues. Even with appropriate methodologies and frameworks, mismatched component assumptions still need to be addressed properly. They will however more likely be identified earlier in the design phase [4].

The fact that autonomous mobile robotics is still a young research field is the third source of mismatched assumptions. Domain representations and system component requirements are not fixed yet, standards are not stabilized and the field needs to adapt to new development strategies and new technologies for an undetermined period of time [10], [13], [14]. This instability creates a propitious environment for mismatched assumptions.

These three sources of mismatched assumptions highlight the importance of considering architectural mismatch as an important integration issue when designing a component-based architecture in robotics.

III. DESIGN SOLUTIONS WITH MARIE

MARIE [10], [15] is our distributed component-based middleware framework oriented towards integration of new and existing software for robotics. To address the integration issues related to architectural mismatch, we developed multiple design solutions to obtain a flexible framework that can be adapted to different mismatch scenarios.

With MARIE, we use various solutions inspired by the mismatch repair techniques, i.e., wrappers, bridges and me-
diators. All of these solutions assume that the original components are integrated together without modification (i.e., code is added around components, but their internals are not modified). The following subsections present our design solutions.

![Diagram of Mediator Interoperability Layer (MIL)](image)

**Fig. 1.** Mediator Interoperability Layer (MIL)

### A. Applications Mediation Approach

To implement distributed robotic systems using heterogeneous applications, we adapted the *Mediator* design pattern [16] to create a Mediator Interoperability Layer (MIL), illustrated in Fig 1. The *Mediator* design pattern primarily creates a centralized control unit (named Mediator) interacting with each component independently, and coordinates global interactions between applications to build the desired system. In MARIE, the MIL acts just like the Mediator design pattern, but is implemented as a virtual space where applications can interact together using a common language (similar to the relation between Internet and HTML for example). Note that the use of a virtual space implies that there is no single implementation class of the Mediator, as represented in the original pattern. The Mediator is distributed between all the applications that are linked together through the MIL, decentralizing the MIL’s functionalities and responsibilities.

With the mediation approach, it is possible to create bridges between incompatible applications by having specialized code adapting each of them through the MIL. This way, each application can have its own communication protocols and mechanisms, as long as the MIL supports them and can bridge the application with others. For the robotics community, this approach offers a way to exploit the diversity of communication protocols and mechanisms, to benefit from their strengths and maximize their usage, and to overcome the lack of standards for robotic software system design.

### B. Application Adapter

Existing applications do not necessarily implement the mechanisms, expose the interfaces or use a communication protocol that would make them compatible with the MIL. Changing an application’s code to add the required functionalities is a solution we try to avoid whenever possible. Instead, we use the wrapper repair technique to create a component which is compatible with the MIL, extending the application’s functionalities without direct modifications. The main role of the wrapper component is to translate application service interfaces to make them compatible with the MIL’s interface. In MARIE, wrapper components used to encapsulate applications are called Application Adapters (AA).

To create an AA or any other component, MARIE offers a development framework called the Component Framework, illustrated in Fig. 2. The *Handler* is responsible for the translation between application interfaces and the MIL interface. *Ports*, explained in Section III-C, are used to communicate with other components through the MIL.

One important design issue when creating an AA is to decide how to offer application services to other components through the MIL. The wrapper repair technique allows to partially or completely modify the application interface exposed to other components via the MIL. It is up to the AA developer to choose the right strategy depending on the system to develop. In some cases, the wrapper can play an important role in avoiding component mismatches by adapting the application’s interface directly to a non-conflicting AA interface. Moreover, one application can be wrapped by multiple AAs, offering different interfaces to fit specific mismatch requirements.

### C. Ports

The mediation approach makes it possible to create bridges between incompatible applications. The first step in creating such bridges is to wrap applications in AAs. Then, to communicate together through the MIL, AAs need to use common communication mechanisms (e.g., communication interfaces, communication protocols and communication languages). This is realized using Ports, as illustrated in Fig. 3.

Operations and processing algorithms of an application are often designed independently of how data are received or sent. Component functionalities can therefore be separated from the communication mechanisms. Ports are designed accordingly, and allow developers to defer the selection of communication mechanisms (which depend on the components that need to be interconnected) to the integration phase or even to runtime. Mismatches related to the nature of the connectors can be solved simply by bridging two or more components together without affecting AA implementations directly. In MARIE, a Communication Strategy (CS) module is responsible for implementing and handling communication protocols (socket TCP/IP, socket UDP, Shared Memory, CORBA, IPC, COM, etc). The Strategy design pattern [16] is used by a CS to define a set of interchangeable communication mechanisms and to create a loosely coupled relation
between the CS’s clients and implementation details of the communication mechanisms.

Another bridge mechanism offered by Ports is the possibility to execute data processing algorithms (filters, converters, etc.) on data before sending it to a component or through the MIL. The processing algorithms are encapsulated in a Cascading Functional Block (CFB). A CFB applies a processing algorithm before passing resulting data to another processing block or any other component (similar to the Pipe-And-Filter architectural pattern [17]). This bridge mechanism enables correction of integration issues between components that do not share the same data representation. Decoupling these processing algorithms from AA functionalities facilitates mismatch repairs by avoiding direct modification of AA implementations. In addition, CFBs can be designed to be easily reused by any component.

**D. Data, DataFactory & Serializer/Deserializer**

Data representation is a common source of mismatch when applications are developed independently using different standards. Most of the time, fixing mismatch issues relies on either changing one application data representation to fit the others, or to support as many data representations as required and doing conversion when necessary. In MARIE, we adopted the second strategy.

MARIE offers two data formats to represent data types: class format and serialized format. In the class format, custom data types are subclasses of a base class named `DataAbstract`. In the serialized format, a sequence of characters is determined according to the communication protocol it relies on.

Converting data from one format to another is a frequent operation depending on how data is used by components. MARIE offers Serializers/Deserializers (SerDes) to support conversion from serialized format to class format, and vice versa, for a specific communication protocol. MARIE also offers what is called a DataFactory, which is a composition of SerDes (one for each data type supported in the DataFactory) for a specific communication protocol. There are as many DataFactory as there are supported communication protocols.

With this design, adding support for new data types is only a matter of creating a subclass for this new data type, and to create as many SerDes as required to support the different serialization formats (e.g., different communication protocols). Ports, CFBs and CSs are not affected by the presence of new data types as they are always using their abstract forms (DataAbstract base class for class format, and sequence of characters in the serialized format).

**E. Communication Adapter**

There is another type of component in MARIE called Communication Adapter (CA) (also based on the Component Framework illustrated in Fig. 2). Communication adapters are specialized components used to fix communication mismatches between AAs. As explained in Section III-B, when designing an AA, the designer must choose how an application service interface is offered to other components via the MIL. These design choices may lead to communication mismatches between two AA MIL’s interfaces, preventing interoperability. Instead of changing the AA interfaces, a CA can be introduced in the system to provide interoperability between components. Fig. 4 presents two scenarios in which CAs are useful.

The first scenario involves AA #1, which needs to send data to AA #2 and AA #3. Instead of adding an additional output to the AA #1 interface or using a communication mechanism allowing AA #1’s Port to handle multiple connections, introducing a Splitter CA can be a better alternative. The Splitter CA is designed to be configured with a variable number of inputs and outputs, and its only function is to forward input data to its outputs. With this solution, no modification is required to the various AAs, and the Splitter CA can be reused in other similar situations.

The second scenario involves AA #4, which needs to exchange data with AA #5. Unfortunately, AA #4 is designed to send data using a synchronous push communication mechanism, and AA #5 is designed to fetch data using an asynchronous pull communication mechanism which creates an incompatibility. Here again, instead of modifying the AA interfaces, introducing a Mailbox CA solves the communication mismatch. The Mailbox CA in this scenario is implemented as a buffer that can store incoming data coming from AA #4, and pop the data from the stack when receiving a pull data enquiry from AA #5.

**IV. ADDRESSING ARCHITECTURAL MISMATCHES IN THE DESIGN OF A SOCIALLY INTERACTIVE AUTONOMOUS MOBILE ROBOT**

Spartacus [18] is a socially interactive mobile robot designed to operate in real life settings. The robot has to navigate and localize autonomously, extract visual information from the world (such as reading messages, tracking
people), localize, track and separate sound sources for enhanced speech recognition and dialogue interaction, provide graphical information through its touch screen interface, and schedule tasks on its own.

To illustrate how MARIE contributed in handling architectural mismatches, Fig. 5 represents a small portion of Spartacus’ software architecture created with MARIE. It covers sensing and acting in simulation and real robot setups, localization, path planning, and part of the computational architecture [18] responsible for the robot’s navigation, reasoning and interaction capabilities. Using the rapid-prototyping design approach, we implemented this portion of the design by integrating components gradually and repairing the mismatches as they were detected.

Our first concern was with data representation, knowing that multiple heterogenous applications had to be integrated together. We decided to create a class data library to be used by all AAs, and to do conversions when required. We also needed a serialized data format to distribute applications on multiple computing nodes. We opted for a custom-designed XML protocol implemented by robust and freely available libraries. Having an easy to read data representation also facilitated debugging. On the other hand, we knew that XML is not particularity optimal for communications as it produces large amounts of characters to represent data, which means that we would probably need to support another protocol to speed up communications. To avoid writing code in each AA for our XML DataFactory, we created two CFBs, one for XML serialization and the other for XML deserialization, which we instantiated in every Port. This way, we could eventually change these CF Bs to use another protocol without affecting AA implementations directly.

Another concern was the communication mechanisms used in our setup. We developed two CS: Socket TCP/IP CS to be able to distribute applications on multiple computing nodes, and Shared Memory CS for efficient data exchanges between AAs on the same computing node. At the time we made these choices, we were not sure whether or not they would be sufficient for all communication requirements of the global system, but they fit our needs for the first exploration and integration phase. New CSs could be added later with little impact on the overall design.

The first integrated application was Player [19], which is specialized for sensor and actuator abstraction, making it possible to seamlessly switch between a simulated environment and the real world environment. To provide this capability, Player interfaces two simulation environments, Stage (2D) and Gazebo (3D), and many robots and sensor devices used by robotic developers. Player AA is a simple wrapper around Player’s API to receive motor commands, and output laser and odometry data at a fixed frequency (10 Hz). For the other drivers more specific to Spartacus’ implementation and that were not available in Player (e.g., odometry), we decided to encapsulate our driver in another AA, Spartacus AA, that reproduces the Player AA interfaces. It would have taken substantial time and effort to convert our current drivers in Player’s API requirements. With this strategy, we could use Player AA for simulation and switch to Spartacus AA for the real environment setup, while keeping the Player AA for our laser range-finder in both settings. The last step was to add a third type of CA, Switch CA, in order to switch odometry output from the simulation to the real settings, and vice versa. The gyroscope interface was added later, after observing that wheel odometry was not accurate enough for Spartacus’ intended use. We could not add the gyroscope driver in Spartacus AA, due to the nature of their respective communication mechanisms. Therefore, we simply decided to create an independent AA for the gyroscope, Gyros AA, and to modify Spartacus AA by adding a new input to it.

The second integrated application was CARMEN, the Carnegie Mellon navigation toolkit [20]. CARMEN is a software package for laser-based autonomous navigation. Wrapping this application in an AA was a little bit trickier compared to Player, as it is composed of small executables communicating through a central server. Moreover, CARMEN’s default mode was not exactly what was required by our computational architecture. We decided to create two different AAs, CARMEN Localizer AA and CARMEN Path Planner AA, each of which interfacing specific parts of CARMEN’s functionalities, and let the computational architecture process resulting information independently. Interconnecting these AAs with Player AA was quite straightforward as we designed CARMEN’s AA to be compatible. We only needed to add a Splitter CA to forward laser data to both CARMEN’s AAs.

The last integrated application is FlowDesigner/RobotFlow (FD) [21], a modular data-flow programming environment that allows, through its graphical user interface, to easily connect reusable software blocks. This application was used to implement Spartacus’ computational architecture. The resulting AA, Behavior & Arbitration FD AA (FD AA), asynchronously fetches data from CARMEN’s AAs and Player AA, processes the information, and produces motor commands at a fixed rate (5 Hz) according to the current tasks to achieve. Integrating FD AA in the setup required to add a Splitter CA to receive the appropriate data and to add Mailbox CAs for each input, as the data arrives using a synchronous push communication mechanism and FD AA fetches data using an asynchronous pull communication mechanism.
V. Discussion

Detecting architectural mismatches is surely one important step in order to solve integration issues. As explained in section II, avoiding mismatched assumptions can be difficult even with good design practices and systemic interoperability analysis. Through our experiments with MARIE, we observed that it provides a flexible component-based software architecture for quick component integration in building complete systems, and it can be used to rapidly evaluate software designs and component implementations and integration within a complete system. Even with the potential drawback of adding computational overhead to repair mismatches (which is not covered in this paper), we definitely gained on system integration capabilities that otherwise would possibly have required to reimplementation all applications to be intrinsically compatible. This way, we were able to discover integration issues sooner in the development cycle, which had an important impact on our global system development strategies.

On the other hand, working with a complete system introduced a level of complexity that contributes to create a setup which is harder to analyze and debug. Even with good software tools like MARIE, we observed that dealing with such a level of integration when building a robotics system requires adapted development methodologies and documentation to accelerate development, in order to spend more time solving robotics challenges.

VI. Conclusion

This paper demonstrates the importance of considering architectural mismatch when using a component-based development approach for software integration in autonomous mobile robotics. Integrating heterogeneous software typically developed independently, under no broadly adopted standards, methodologies and frameworks, almost inevitably leads to integration issues related to architectural mismatches. MARIE offers a component-based software architecture that can address these issues by implementing multiple design solutions to repair architectural mismatches. Our experience with MARIE in the design of a socially interactive autonomous mobile robot shows very encouraging results, and illustrates how such a tool can contribute to a better understanding of the underlying requirements of robotic software design challenges.

Future work includes developing a structured development methodology based on MARIE integration capabilities to avoid, detect and repair architectural mismatches, and on a more in-depth study of the impact of overhead introduced by MARIE when developing and deploying systems. MARIE is available as open source at http://marie.sf.net.

VII. Acknowledgements

The authors gratefully acknowledge the contribution of the Canada Research Chair (CRC), the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Canadian Foundation for Innovation (CFI), in the support of this work.

References