A Model-Based Approach to Implementing the Arrowhead Framework for Industrial IoT

Jan van Deventer

October 31, 2024

Abstract

The increasing complexity of Industrial Internet of Things (IoT) systems necessitates robust methodologies for efficient development and deployment. This paper presents a model-based implementation of an IoT solution to swiftly empower engineers to program compliant systems and technician to deploy these systems. This paper contributes to the field of IoT by demonstrating how model based system engineering can streamline the development and deployment process, ultimately improving system maintainability and scalability.

For illustrative purposes, the Arrowhead framework, an IoT automation solution, is used along with a climate control use case. This work highlights key aspects such as resilience, interoperability, cybersecurity, and semantic reasoning in IoT solutions.

Keywords

MBSE, CPS, OPC UA, Modbus, MQTT, Golang

1 Introduction

Once upon a time, there were telephone directories. Their names were White Pages and Yellow Pages. They were quite thick soft cover books, usually published on a yearly basis. The White Pages listed the telephone subscribers from a specific area in alphabetical order with their address and telephone number. The Yellow Pages listed businesses in alphabetical order by the type of the services they offered. So, if anybody had a clogged kitchen sink, their fingers would just have to walk across the Yellow Pages to find a suitable plumber. Once contacted, the plumber would, with a smile, come to clear the drain. It should be noted that the Yellow Pages only exposed the registered services, such as those offered by a plumber or a plumbing company. The plumber himself is not a service but a resource, with an address and a phone number. It is the plumber's skills that are exposed as services.

The Internet has revolutionized our society and abolished the White and Yellow Pages. People nowadays swipe over their Internet-connected devices to find the services they seek, and have forgotten about the Yellow Pages. Nonetheless, the Internet has not overcome the plumber because it has not succeeded in digitalizing him (though it has done so with bank tellers, who are not part of this odd fairytale).

As computing devices have shrunk in size and cost while increasing in power, their impact has extended beyond society into industry. The Internet Revolution began in the late 20th century, gaining momentum with the world wide web; soon industries adopted it to enable communication between devices and systems. This expansion has fueled movements like Germany's Industry 4.0 and the Industrial Internet Consortium (IIC) in the United States. These movements are part of the so-called fourth industrial revolution, which democratizes how "things" communicate, forming the Internet of Things (IoT).

The rise of IoT has disrupted traditional industrial structures, turning the hierarchical ISA-95 pyramid into a complex web of interconnected systems. To

manage the potential chaos, various reference architectures and frameworks have been proposed. Examples include RAMI 4.0 for Industry 4.0 and the Industrial Internet Reference Architecture (IIRA), which provide blueprints for building interoperable systems. In addition, proprietary architectures such as AWS IoT and Microsoft Azure IoT are also prevalent. European frameworks, like FIWARE and Arrowhead, offer additional alternatives, especially within European-funded initiatives.

These frameworks provide valuable guidance, but their implementation is not straightforward. Developers must transform abstract models into operational systems, which is both complex and time-consuming. This work focuses on how Model-Based Systems Engineering (MBSE) can streamline this process.

The hypothesis of this work is that MBSE can expedite the development process by providing clear models that simplify the transformation from framework to code. By doing so, MBSE can enhance knowledge transfer to technicians and engineers, making the systems more resilient through better understanding.

Several studies have highlighted the importance of training in the successful deployment of IoT solutions. For instance, Valmohammadi emphasizes the need for investment in IT infrastructure and employee training to accelerate IoT adoption in organizations. The steep learning curve of IoT systems requires a structured approach, where comprehensive educational programs are crucial to equipping engineers with the necessary skills.

Model-Driven Engineering (MDE), a subset of MBSE, has been proposed as a solution to increase exciplicitness of concepts in IoT development. By automating code generation for hardware devices and server-side applications, MDE helps ease the learning curve and reduce the time required for engineers to develop IoT systems.

The goal of this work is to enable rapid training for deployment technicians within half a day and for application engineers within one day. After reading this article, the reader should reflect on which IoT implementations they are most comfortable with and why.

To demonstrate this concept, MBSE is applied to the Arrowhead framework, with its implementation relying on the Google Go programming language. The implementation is made available as an open-source project on GitHub. In the following sections, the Arrowhead framework is briefly introduced, followed by a model of the framework that is packaged into a Go module. A climate control use case is presented to illustrate a local cloud deployment. Finally, a discussion covers important topics such as resilience, interoperability, compatibility, cybersecurity, and semantic reasoning.

2 The Arrowhead Framework

The Arrowhead framework was developed to address the needs of the European manufacturing industry. It was funded through various European Union projects aimed at improving cybersecurity and interoperability between legacy, current, and future automation systems. The framework's primary objective is to provide a low-latency, service-oriented architecture where systems can communicate directly with one another. To secure this communication, the Arrowhead framework employs encryption mechanisms based on public key infrastructure.

In the context of IoT, each entity consists of a "thing" and an interfacing software that connects the thing to the cyber world and the Internet. The "thing" can be a physical device like a sensor or actuator, or it can be a more abstract asset such as an algorithm or database. In RAMI 4.0 terms, the thing is called an asset, and the software that interacts with it is called a shell. The Arrowhead framework combines the software and asset into what it calls a system. In this work, we use the terms "husk" for the software interface and "unit asset" for the underlying resource. This distinction helps highlight the differences between RAMI 4.0 and the Arrowhead framework in how they model asset administration and services.

The Arrowhead framework adopts a service-oriented architecture (SOA) sim-

ilar to the Yellow Pages analogy introduced earlier. Just as the Yellow Pages register available services, the Arrowhead framework's Service Registry system tracks services offered by different systems. Each system exposes its unit asset's functionalities as services, which other systems can discover and consume at run time. Late binding allows systems to find and connect with each other dynamically during their operation, ensuring flexibility and adaptability.

The framework provides three core systems to manage the service lifecycle: the Service Registry, Orchestrator, and Authorization systems. When a system seeks a service, it communicates with the Orchestrator, which checks the Service Registry for available services and consults the Authorization system to verify that the service can be consumed. The Orchestrator then provides the requesting system with the service's resource location, allowing it to consume the service directly.

One of the key features of the Arrowhead framework is its fine-grained service authorization mechanism. For example, a data logger may be authorized to access the state of an actuator but not to modify it, while a controller system may have permission to both read and write actuator states.

A distinctive aspect of the Arrowhead framework is the concept of a local cloud, which is a set of systems that function independently but can communicate with other clouds when necessary. Each local cloud must have a Service Registry, Orchestrator, and Authorization system to maintain autonomy and security. While the size of a local cloud is not predefined, it should remain manageable to ensure efficient operation. Communication between clouds is facilitated by GateKeeper and Gateway systems, allowing for secure, cross-cloud interactions.

The Arrowhead framework addresses the challenges of interoperability through the husk (software interface). On the one hand, the husk communicates with the unit asset, typically using protocols defined at design time. On the other hand, it facilitates communication between systems using various protocols, with protocol-specific servers handling different message formats and payloads.

With the overall concepts of the Arrowhead framework presented, the next step is to model this understanding. In the following section, we will demonstrate how the framework can be modeled using both structural and behavioral diagrams, facilitating a clearer understanding of its components and interactions.

3 Modeling the Framework

Visual modeling languages are used to provide a standardized way to visualize the design and structure of systems. Examples of these languages include the Universal Modeling Language (UML) and the Systems Modeling Language (SysML). These languages offer a taxonomy of diagrams that can be broadly divided into two categories: structural and behavioral diagrams. Both categories are necessary to fully describe the systems and interactions in a software architecture, and they are applied in this work.

To begin, the structural aspects of the Arrowhead framework need to be modeled. A local cloud is composed of systems that run on devices, each system consisting of a husk and one or more unit assets. The husk is responsible for exposing the functionalities of the unit assets as services that other systems can consume. Figure 1 presents a symbolic aggregation of these components as a diaagrams inspired from RAMI 4.0 and Arrowhead documentation and as a class or block body diagram. These high-level models emphasize the relationships between the systems, unit assets, and services. In the top figure, the lollypop and socket extensions represent providing and consuming services respectively. The Arrowhead framework usually depicts a system as a yellow block with its lollypop and socket extensions. The lower diagram represents a class or block body diagram and can be used as a map when navigating the source code.

Next, the behavioral aspects of the Arrowhead framework are modeled to capture the dynamic interactions between systems. Figure 2 illustrates the main use cases for a generic Arrowhead system. With the word generic is meant

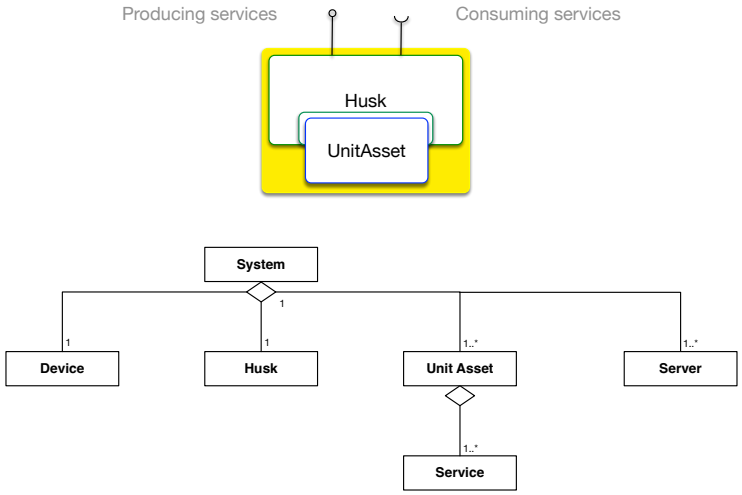

Figure 1: Two representations of a system. RAMI 4.0-Arrowhead framework style representation, and class or block body diagram.

here that all Arrowhead systems make use of these use cases. To the left, the deployment technician is the only human actor in the figure and plays a crucial role in configuring the system, which does not require recompiling the software. The same configuration use case is used for all systems, even future ones. The other actors are systems within the same local cloud.

One key framework stipulation is that all systems must register their services with the leading Service Registrar. Even the leading Service Registrar must register its services with itself to ensure consistency and resiliency within the local cloud. Service discovery is managed by the Orchestrator, which checks for available services and verifies authorization before presenting the requesting system with the appropriate resource location. The Authorization use case is grayed out because it has not yet been completed in the current implementation.

To the right of Figure 2, the interaction between the system (or husk) and its unit asset(s) is depicted. This relationship is clear when dealing with physical unit assets such as sensors, actuators, or PLCs, where the system interfaces with these external devices. In cases where the unit asset is an algorithm, the interchange may remain hidden within the system, and wraps the intellectual property, which is then not exposed.

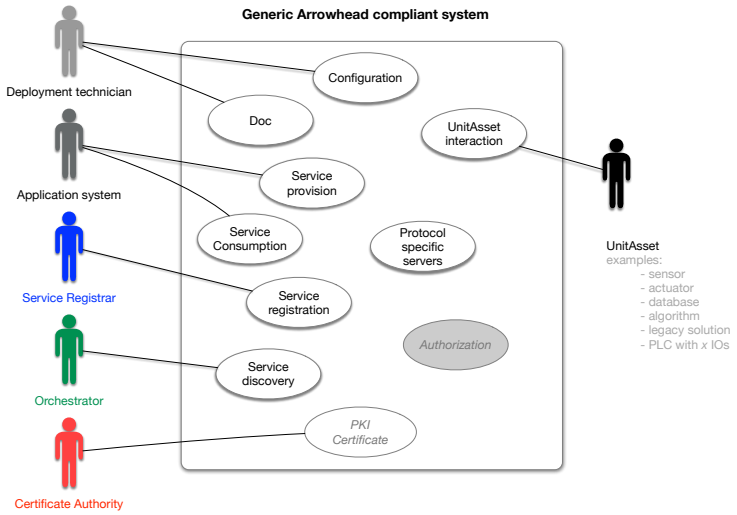

Figure 2: Use cases for a generic Arrowhead-compliant system, showing interactions within a local cloud.

Using models to represent both the structural and behavioral aspects of the framework facilitates communication among stakeholders. These models help ensure a shared understanding of the system's architecture and simplify the software implementation process by offering a visual guide to the system's components and interactions.

4 The Implementation

Implementing the concepts of a reference architecture or framework, even when well-modeled, can be challenging. It is during the implementation that subtle details reveal their true influence, often complicating the process.

One of the core strengths of Model-Based Systems Engineering (MBSE) is its independence from specific reference architectures, frameworks, or programming languages. In other words, MBSE can and should be applied to any software solution if the hypothesis about its effectiveness holds true.

In this particular work, the chosen programming language for implementing the framework is Go (Golang). The implementation is made available through two GitHub repositories: one for the library, called `mbaigo`, and another for a

collection of systems, called `systems`. The `mbaigo` library is composed of three packages: components, forms, and use cases.

The `components` package contains the core entities such as system, device, husk, unit asset, and service, which were introduced in the previous section. The `use cases` package corresponds to the primary use cases of the Arrowhead framework, such as service registration, discovery, and consumption. Finally, the `forms` package handles the information exchange payloads.

The `systems` repository contains several example systems that demonstrate how to use the `mbaigo` library. Each system in `systems` is organized into two files: `systemname.go` and `thing.go`. This separation is intended for pedagogical reasons, where `systemname.go` deals with the cyber aspect of the system, and `thing.go` deals with the unit asset.

The main function of each system aggregates components of the `mbaigo` library and starts the system. Figure 3 shows the activity diagram for the `main()` function. The system is first instantiated and configured using a configuration file, which is located in the same directory as the executable. If the configuration file is missing, it is generated during the system's first run, and the system shuts down to allow the deployment technician to update the configuration.

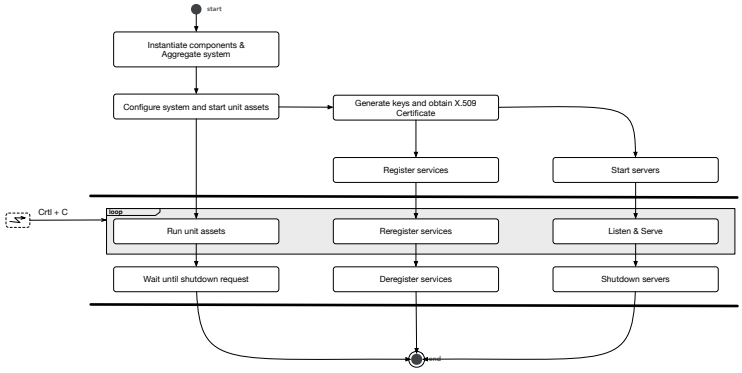

Figure 3: Activity diagram of the `main()` function of a generic system. The gray looping block consists of processes that are concurrent goroutines, which terminate upon an external request.

The `thing.go` file defines the unit asset structure and specifies which of its attributes are configurable. Once the configuration file is updated, the unit

assets are instantiated and started with the appropriate settings. For example, in the case of a 1-wire temperature sensor, the configuration file would specify the serial number of each sensor and its respective location.

The system generates a private key and posts a certificate signing request to the Certificate Authority. These private keys are not stored on the file system and are lost upon system shutdown along with the certificates. When certificates are use for secured communication, mutual certification is required, that is both the client and the server must have a valid certificate. At the time of writing, the authenticating system is missing so we cannot claim complete cybersecurity. Both software and hardware authentication is under development.

After configuration and security setups, the services are registered with the leading Service Registrar. This registration process repeats at regular intervals to maintain an up-to-date catalog of available services with the leading Service Registrar. If the system fails to reregister its services, the Service Registrar removes the entries to prevent stale information. These service re-registrations are, like the unit assets' operations, concurrent processes that rely on lightweight goroutines. Finally, the system starts its servers and listens for incoming requests, remaining in operation until it receives a shutdown command.

When a shutdown command is received, the system deregisters all its services, shuts down the servers, and disconnects from its unit assets. This structured shutdown process ensures a graceful termination of the system.

Having having an http server, each system supplies a web server to empower the deployment technician to verify the installation with a standard web browser. The web pages present information about the system, its hosting device, its unit assets and their services. The address to these web pages is provided by the system upon start up.

5 The Prototypes

Prototypes play a critical role in developing implementations and facilitating knowledge transfer. In this work, several prototype systems are introduced, five of which are used to demonstrate the concept of a local cloud. The other systems include as assets a camera, a microphone, an OPC UA server, a Modbus server, and an MQTT broker providing further examples of how different assets are seamlessly integrated into the framework.

The local cloud prototype focuses on a climate control use case, making it relatable to the reader. This cloud consists of five systems: a Service Registrar, an Orchestrator, a 1-wire temperature sensor, a pulse-width modulated servo motor emulating a valve, and a thermostat. Each system registers its services with the Service Registrar, ensuring they are available for discovery and consumption by other systems.

The thermostat system communicates with the Orchestrator to discover the temperature sensor and valve services within the same room. The Orchestrator checks the Service Registry to confirm the availability of these services and provides the thermostat with their URLs. Every ten seconds, the thermostat makes a request to the temperature sensor to obtain the current temperature and calculates a new valve position. It then makes a request to the servo motor to update the valve position based on the calculated value.

Besides the position service, the thermostat system also offers two additional services: a jitter service and a set point service. The jitter service reports the time taken to complete the two requests, which is consistently under ten milliseconds. The set point service allows users to read or update the thermostat's desired temperature. With such a setup, one can easily have a low cost demand response solution to save energy [**?**]

The camera system, with a Raspberry Pi camera as an asset, takes a picture as a service upon receiving a request. However, instead of returning the image directly, the system responds with a JSON payload containing the location of

the image and a timestamp. A second request is required to retrieve the image file as a JPEG, illustrating how files can be exchanged as part of a service. The recorder system is similar to provide sound recordings. It uses a USB connected microphone as an asset.

The PLC systems provide access to input/output signals and registers, depending on the specific communication protocol used. The UAclient system, with an OPC UA server as an asset, offers `browse` and `access` services to nodes. By default the object folder node is presented allowing a deployment technician to navigate the available nodes and select the ones of interest. The Modbus system, on the other hand, requires prior knowledge of the PLC's register map, which is defined in the system's configuration file. Another popular IoT protocol is MQTT that relies on a broker with a publish-subscribe paradigm of topics. The Telegrapher system has an MQTT broker as an assets and metamorphose topic into services.

Readers are encouraged to explore the GitHub repositories to better understand the implementation of these prototypes. Starting with simpler systems like the temperature sensor or camera provides a good foundation before examining more complex systems such as the Service Registrar. The Service Registrar system is the most intricate, as it includes both an embedded SQL database and a scheduler that handles the automatic cleanup of expired service registrations.

The presented prototypes demonstrate how the Arrowhead framework can be implemented and deployed in a real-world scenario. In the next section, we discuss key aspects such as resilience, interoperability, cybersecurity, and the potential for reasoning in a local cloud environment.

6 Discussion

The goal of this work is to facilitate knowledge transfer through the use of Model-Based Systems Engineering. The reader should now have a comprehensive understanding of the current implementation of the IoT framework, as

guided by the model diagrams and repository examples. Using MBSE to engineer both the structure and behavior of systems supports this goal while offering additional benefits that are not immediately obvious.

The fairytale analogy used in the introduction has a deeper purpose here. Abstraction in modeling provides significant advantages, but relating it to something familiar—like the Yellow Pages—helps ease the cognitive load. The example of the plumber also serves to illustrate the relationship between a system, its assets, and the services it exposes. This reflection promotes a deeper understanding of how services and resources interact in an IoT framework.

One significant advantage of a service-oriented architecture, such as the one used in the Arrowhead framework, is that the systems are loosely coupled. This means that systems can be developed independently and bound together only at run time. For example, the temperature sensor system provides its temperature service without needing to know how it will be used by other systems. Similarly, the thermostat system does not need to know whether the temperature reading comes from a 1-wire system or a PLC, as long as the service matches its requirements (service definition, unit, and location).

Interestingly, the use cases implemented in the `mbaigo` library are also loosely coupled. Each use case is generally independent of the others and is only tied to the components and form packages. This allows continuous development and evolution of each use case with minimal or no impact on the overall functionality of the systems. As a result, the framework can evolve over time without requiring major redesigns.

One of the key requirements of the Arrowhead framework is interoperability. The presented prototypes demonstrate that services can be consumed regardless of whether the asset uses 1-wire, OPC UA, Modbus, or MQTT. Additionally, interoperability between systems is also crucial, especially when dealing with constrained devices that may rely on protocols like CoAP. To address this, the `mbaigo` library provides protocol-specific servers, allowing seamless integration of various communication protocols.

Interoperability extends to the payload formats as well. The payloads exchanged between systems are versioned to ensure backward compatibility, meaning that older systems can still communicate with newer ones even if the data schema evolves. This is achieved by including a version field in the forms, allowing the correct mapping of information to the appropriate object attributes. More technically, Forms is really a Go interface that each form must implement. The mbaigo library offers packing and unpacking functions that handle the serialization (xml and json) and marshaling to the correct form.

In the use case diagram, the deployment technician is the only human who interacts directly with a generic system. The systems provide their `systemconfig.json` at their initial start up. It is the same configuration use case and code used for all systems making configuration of systems consistent.

Efficiency is paramount in the development, deployment, and maintenance of IoT systems. MBSE supports maintenance by ensuring that bugs in use cases only need to be fixed once, as the use cases are shared across systems. Another area that benefits from efficiency is security, particularly when using public key infrastructure. Automating the generation of keys and certificates at startup reduces the cost and complexity of deploying secure systems, preventing security vulnerabilities.

In an industrial plant, systems and hardware can change or fail. To address failures, the Arrowhead framework requires systems to reregister their services at regular intervals, ensuring that the Service Registry maintains an up-to-date list of available services. If a system fails to reregister, its services are automatically deregistered, preventing stale entries. If the leading Service Registrar fails, another one takes over the leading role, if more than Service Registrar is deployed in a local cloud. Resiliency is a basic requirement of the implementation.

The clearly defined data models (such as those in the `components` package) lay the foundation for information models and reasoning. For example, one can reason that a local cloud is non-functional without a Service Registrar, but

it becomes resilient with multiple registrars. These reasoning capabilities are part of future work, where the Arrowhead framework ontology will be further explored. The semantic model of deployed systems is already accessible similar to the online documentation (using `onto` instead of `doc` in the URL path).

Given the emphasis on knowledge transfer, three types of documentation are provided. First, manually written `README.md` files are included in the repositories to explain the systems and their purpose. Second, an online "black box" documentation is available through the system's web server, allowing deployment technicians to verify system configurations via a standard web browser. Third, a "white box" documentation is generated using Go Doc, which provides detailed documentation accessible through a supported integrated development environement or the `pkg.go.dev` website.

The choice of Google Go as the implementation language offers several advantages. Its straightforward cross-compilation process minimizes platform dependencies, providing compatibility across various systems while yielding fast executables. Additionally, Go's lightweight goroutines and channels make concurrent programming more manageable, allowing multiple services (e.g., service registration) to operate independently within an efficient concurrency model. The resulting system binaries are typically lightweight, often around 10 MB in size, depending on the application's complexity.

In summary, MBSE has proven to be an effective method for developing and deploying an IoT framework while simplifying knowledge transfer and system maintenance. In the next section, we conclude the work and reflect on the broader implications of this approach for IoT development.

7 Conclusion

The application of MBSE to the Arrowhead framework has yielded several benefits. First, the use of models has facilitated knowledge transfer by providing a clear, visual guide to the system's architecture and behavior. Second, the

separation of concerns between systems and their unit assets has allowed for greater flexibility in configuration, without the need for recompilation. Finally, the service-oriented architecture employed by Arrowhead framework, combined with the loosely coupled use cases in the `mbaigo` library, has supported a scalable and resilient system design.

Readers are encouraged to explore the GitHub repositories associated with this work, where the complete implementation of the Arrowhead framework in Google Go is available. By studying the examples provided and experimenting with different systems, practitioners can deepen their understanding of how to apply MBSE in real-world scenarios.

The implications of this work extend beyond the specific implementation presented here. The MBSE approach can be applied to other IoT frameworks and architectures, enabling faster development, better knowledge transfer, and more maintainable systems across various industrial and commercial contexts. While the Arrowhead framework and IoT systems in general can be complex, this work demonstrates that such complexity does not necessarily translate into complicated implementations.