34th Annual INCOSE international symposium
hybrid event
Dublin, Ireland
July 2 - 6, 2024

# Modeling Enterprise Software with UAF

Matthew Hause
SSI
3208 Misty Oaks Way
Matthew.Hause@Hotmail.com

Lars-Olof Kihlström
CAG Syntell
P.O.Box 10022, SE-10055 Stockholm, Sweden
lars.olof.kihlstrom@cag.se



**Abstract**. Systems and Software Engineers often have an uneasy relationship. The job of the systems engineer is to work with the stakeholders to define a set of requirements that meet their needs. These are then allocated to various solution spaces such as electronic hardware, mechanical, procedural, and software among others. For many systems, the functional requirements are almost exclusively software requirements. Correspondingly, as an increasing amount of project manpower, schedule time, and budget are allocated to software, it becomes increasingly important that systems and software engineers communicate effectively. The Systems Modeling Language (SysML) has helped in this regard in that it can provide executable behavioral models with precise semantics to express software requirements in a model. These models define "What is required" without overly constraining the implementation. In addition, SysML can be used to define performance constraints, required concurrency, hardware memory and processor budgets, interfaces, safety critical requirements, etc. These aspects are essential for software engineers to understand the constraints and limitations of their environment. At the System of Systems (SoS)/Enterprise level, defining software/systems employs a similar pattern, but at a higher level of abstraction. In the Unified Architecture Framework, capabilities are defined for the enterprise, with systems and software allocated to realize the capabilities. In the same way that capabilities depend on one another, the implementing systems and software interact to support each other. In the past, enterprise software would be modeled as residing in mainframes in a federated software pattern. Modern software can be modeled throughout the enterprise in a distributed network that can adapt to the changing needs of the enterprise to do load leveling, dynamic and late binding, reconfiguration, and reallocation of hardware resources as necessary. If the domain includes the Industrial Internet of Things (IIOT), then deployment could include edge devices, embedded software, Programmable Logic Controllers (PLC), PCs, servers, cloud computing, and of course mainframes. The Object management Group (OMG) Data Distribution Services (DDS) standard enables these capabilities across these devices in a universal format implemented by multiple vendors. However, before this complex system of systems can be implemented, it must first be architected and designed to ensure that it will be fit for purpose both now and as the complex system of systems expands and evolves. This paper will examine the aspects of modeling software in the UAF, and how it can help guide enterprise and system and software architecture.

**Keywords.** Enterprise Software, UAF, MBSE, Frameworks.

## Introduction

Software systems are unique in their configurability, diversity, and malleability. Software systems can be reconfigured in a matter of moments, whereas hardware systems can take days if not weeks or months.

Reconfiguring an assembly line to build a new vehicle is an enormous undertaking involving physical reconfiguration of complex systems and retraining of employees. With the use of flexible manufacturing via robot assembly, 3D printed parts, and augmented reality assisted assembly, the change can be done in a day. Distributed control systems changed the paradigm from federated systems with computing capabilities residing in a single mainframe to multiple computing devices each performing their necessary functions, computing, and coordinating through networks. Computing control or intelligence could be placed where it was needed resulting in autonomous and safer systems. The Industrial Internet of Things (IIoT) expanded the range of available devices and brought in superfast communication systems such as 5G. Sensors changed from simple measuring devices to complex computing devices pre-processing data and performing controls where necessary. These smart, low power devices further expanded the distribution of intelligence throughout the system. These also enabled the digital twin and advanced simulation. Finally, cloud computing meant that systems could be hosted anywhere necessary, and resources expanded and reduced as and when necessary. This diversity of options meant that analysis of alternatives and system architecture became even more complex, enabling smart cities, virtual systems, autonomous vehicles, smart roads, virtual teams working on virtual projects, and a 24/7/365 globally connected world as well as many other examples. This paper will look at smart factory systems, which utilize virtually every possible type of available computing system elements. More specifically, it will look at the software in these systems and the process of creating the system and software architectures.

## *The Software Problem*

Software design has always been problematic. Stafford (1982) writes "It has long been acknowledged that in computerized projects that the single most awkward item to control is software and its productions mainly because of its dependence on human sources and performance as well as the difficulty of identifying software progress. There are many methods to control the possibility of software excesses, and thus minimize risks, but unfortunately no panacea exists to cure every ill" The problem has not improved over time with late and canceled software projects in the news on a regular basis. Identifying problems early in the development cycle provides a more cost-effective means of lowering development costs. Defining the software architecture will help in this regard as the system/software functionality and other issues can be validated, verified, and tested to ensure a "correct" solution. This is especially important now that software has become so ubiquitous and the available architectures and patterns so many and varied.

## The Software Explosion

Software is an essential part of virtually all systems. One example of the explosion of software is the automotive industry. Conant (2023) recounts that "in 1968, Volkswagen released the first car with an engine computer as part of the Bosch D-Jetronic electronic fuel injection (EFI) system. And by 1990, every car produced in the west had computers as well." Charette (2023) interviewed Manfred Broy, emeritus professor of informatics at Technical University, Munich. He stated that "Once, software was a part of the car. Now, software determines the value of a car," The success of a car depends on its software much more than the mechanical side." Nearly all vehicle innovations by auto manufacturers, are now tied to software." (Charette, 2023)

Charette (2023) continues "Today, high-end cars like the BMW 7-series with advanced technology like advanced driver-assist systems (ADAS) may contain 150 ECUs or more, while pick-up trucks like Ford's F-150 top 150 million lines of code. Even low-end vehicles are quickly approaching 100 Electronic Control Units (ECUs) and 100 million of lines of code as more features are becoming standard." There is also the matter of control and management of this amount of software. VW estimates that only 10% of the software in its vehicles is developed in-house. The other 90% is contributed up to 50 different suppliers. This number of suppliers increases complexity, specifically in verification and validation. A recent Strategy Analytics and Aurora Labs survey of software developers across the automotive supply chain asked how difficult it

was to know when a code change in one ECU affects another. 37% said it was difficult, 31% very difficult, 7% close to impossible, while 16% impossible. Car companies and their suppliers are realizing that they must collaborate more to keep tighter control of data configuration management to keep unintended consequences from occurring due to unanticipated ECU code changes. (Charette, 2023) This is in a single tightly controlled product as opposed to a System of Systems (SoS). Clearly, software architecture has become increasingly more important to understand and plan. Doing so, requires clear communication, allocation of responsibilities, a set process, and set boundaries. This has been the case for a while.

## *The Systems and Software Problem*

Campbell (2004) found that the DoD buys systems, but that software is both a critical enabler and a prominent source of risk (both product and process). He also found that systems engineering practices contribute to software risks if they:

- Prematurely over-constrain software engineering choices.
- Inadequately communicate information, including unknowns and uncertainties, needed for effective software engineering.
- Fail to adequately represent and analyze the implications of software design choices in system-level trade studies.

The conclusion was that "Attempting to fix software engineering problems without rethinking the role of systems engineering may limit any potential for improvement" (Campbell, 2004). To understand the problem better let's first look at the differences between systems and software engineers, how they think, and what they do.

Systems and software engineers look at a problem from their own points of view and have different skillsets. Both need to understand underlying business and support requirements to design solutions. Both draw on techniques and processes from multiple disciplines to solve complex problems. They rely on training and experience in their fields as well as in the application area to implement their solutions. They must collaborate with quality assurance (QA), hardware engineers, human factors, etc. Systems engineers start with a universe of potential solutions, which they must narrow down based on the customer needs and requirements, system context, environment, and constraints of the solution space. Systems engineers perform trade-off analysis of different architectural solutions to a problem and allocate requirements to different engineering domains within that solution, including software. Traceability must be created from the system definition to software and other domains.

The Object Management Group (OMG) Systems Modeling Language (SysML) was created to provide a common language between Systems and Software Engineers who were largely using the Unified Modeling Language (UML) at the time to model software. (Fowler, 1997) Other subject matter experts have also learned SysML resulting in a lingua franca for engineers. (OMG, 2019) (Friedenthal et al, 2011) SysML viewpoints provide information to be allocated to the different engineering domains (e.g., hardware, software, procedural, mechanical etc.). Modeling of any sort involves abstracting the elements pertinent to the area of analysis to provide a multitude of views from different domains. Modeling centered on a particular domain will involve "abstracting the abstraction" to consider the information that is appropriate to the viewpoint and the domain. For example, what portion of the system design will be allocated to software. Additional information, application of standards, constraints, etc., will then be added to the model.

On the other hand, software engineers are limited to creating a solution within the hardware space defined for them and within the performance, reliability, and size constraints defined by others. Software engineers focus on implementing software, often in teams separated from users. Systems engineers work with users, stakeholders, and subject matter experts from different domains. Software and systems engineers need to work with each other to achieve a consistent, coherent solution that achieves the customers goals and can

adapt over time. Software engineering almost entirely involves implementing system functional requirements. SysML behavioral diagrams (activity, use case, state, and sequence) can be used to define the required functionality. Block and package diagrams are used to define functional modules and their relationships as well as the environment in which the software will function. Packages can be used to define software libraries necessary for the implementation. Parametric diagrams can be used to define performance constraints. Finally, requirements will be used throughout both the systems and software models. The concepts defined in one domain can be mapped to the other using the allocation relationship.

## *What Do Software Engineers Need?*

Given the assertion that systems engineers can communicate with software engineers via SysML, what do they need to communicate? Hause & Thom, (2008) described a set of essential elements that can define what is required, what environmental and physical constraints exist, and necessary functionality.

- Initial Requirements and Traceability
- System context
- Use Cases
- System behavior
- State Based Behavior
- Causal Sequences
- Hardware/Software Interfaces
- Data Requirements
- System Constraints (Disk, memory, processor, etc.)
- Performance Requirements
- Systems/Software Traceability
- Communications and Connectivity
- Data Sources and Interchange
- Etc.

## *Systems of Systems (SoS)*

A system of systems (SoS) is a "collection of systems, each capable of independent operation, that interoperate together to achieve additional desired capabilities." (Mitre). Maier (1998) listed five key characteristics of SoS:

- Operational independence of component systems
- Managerial independence of component systems
- Geographical distribution
- Emergent behavior, and
- Evolutionary development processes

Other aspects listed were that they have multiple:

- Levels of stakeholders with mixed and possibly competing interests
- Possibly contradictory, objectives and purpose
- Different, operational priorities with no clear escalation routes
- Lifecycles with elements being implemented asynchronously.
- Owners making independent resourcing decisions.

There are several types of SoS as well:

- Directed where the SoS is created and managed to fulfill specific purposes and constituent systems are subordinate. Air traffic control systems are an example.
- Acknowledged with recognized objectives, a designated manager, and resources. Systems retain their independence and changes are based on cooperative agreements - Defense systems, governments, etc.
- Collaborative where component systems interact voluntarily to fulfill central purposes. Public utilities, Cell phone network, Cable TV, the Energy Grid and supporting systems.
- Virtual with no central management authority or centrally agreed upon purpose and where invisible mechanisms maintain it. (Maier, 1998; Dahmann and Baldwin, 2008, ISO 21839, 2019)

One could argue that the factory system used in this paper does not meet the criteria of an SoS. At the very least is it a directed SoS as it has a single owner and a common goal. Given its distributed nature, it serves as a useful example to illustrate the concepts. In an SoS the various parts of the SoS fulfill their purposes independently and in conjunction with the other parts. All SoSs depend on what can be called "Distributed Intelligence". Most if not all the intelligence, behavior, and functionality in a system is provided by software. The question then becomes how to distribute the intelligence via software functionality.

## *Distributed Intelligence*

When designing a system there are several questions to be answered: What goes where? Where should systems be physically located? Where to place components in relation to one another to minimize adverse effects and/or maximize efficiency? How to distribute control (intelligence) throughout a system to increase reliability and reduce installation costs? Traditional design methodologies involve creating a context diagram with the system intelligence (control system) at the center, with internal elements in the context connected to it, and external elements connected to the internal elements. By modeling the flows, interactions, required behavior, causal sequences, etc., the intelligence/controls could be allocated throughout the system elements (Stafford, 1982). There are various patterns for designing control systems. A distributed control system (DCS) is a computerized control system for a process or plant in which autonomous controllers are distributed throughout the system, but there is no central operator supervisory control. This contrasts with systems that use centralized controllers; either discrete controllers located at a central control room or within a central computer. The DCS concept increases reliability and reduces installation costs by localizing control functions near the process plant, with remote monitoring and supervision. Supervisory Control and Data Acquisition (SCADA) and DCS systems are very similar, but DCS tends to be used on large continuous process plants where high reliability and security are important, and the control room is not geographically remote (Eloranta et al, 2014). A DCS increases reliability as control processing (intelligence) is distributed throughout the system rather than centralized in a single area. The physical proximity of the system elements ensures fast control times by reducing network delays. Figure 1 shows the control hierarchy of a traditional distributed control system.

The physical plant is at the level 0, (the field level) and made up of the different systems that perform the factory functions. These assembly systems will contain the robots, controllers, safety equipment, assembly lines, telemetry, etc. that make up the factory.

- Level 0 contains the field devices such as flow and temperature sensors, and final control elements, such as control valves.
- Level 1 contains the industrialized Input/Output (I/O) modules, and their associated distributed electronic processors.
- Level 2 contains the supervisory computers, which collect information from processor nodes on the system, and provide the operator control screens.
- Level 3 is the production control level, which does not directly control the process, but is concerned with monitoring production and monitoring targets.
- Level 4 is the production scheduling level. (Eloranta et al, 2014)

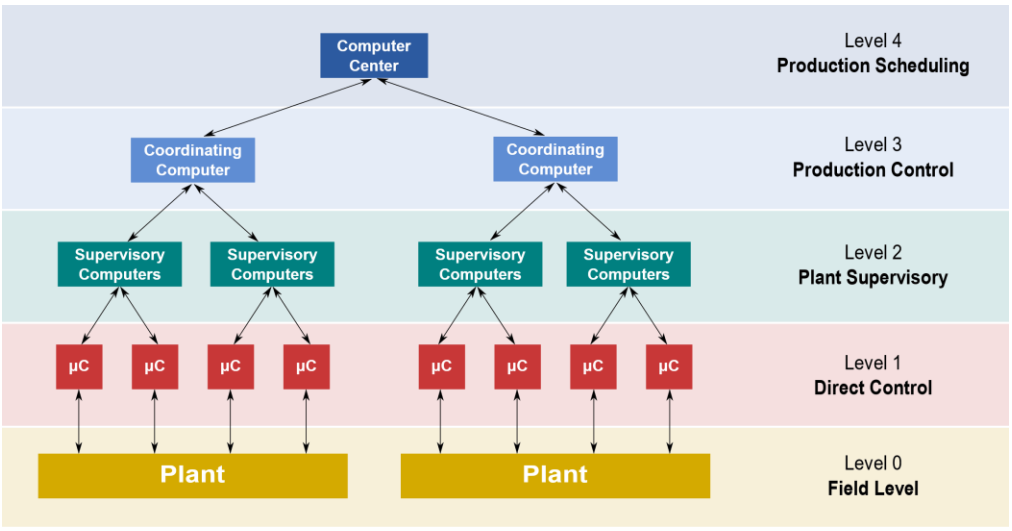


Figure 1. Hierarchical Factory Control System https://commons.wikimedia.org/wiki/File:Functional_levels_of_a_Distributed_Control_System.svg

This standard hierarchical control architecture was standard for much of the history of control systems. Partly this was due to the limitations of the hardware, software, and communications technology at the time. Another influence were the company structures and processes at the time. Conway's law states that "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." (Conway, 1968) Changing process, organizations, and structures also brought change. Reviewing the structure, software functionality at each level must be well defined in terms of required functionality, data, and interfaces. This corresponds to classic Object Oriented (OO) design rules for objects. For each object the developer needs to define its purpose and then determine what functionality does it need (operations), what does it know (attributes), and who does it need to communicate with (associations). Fowler, Scott (1997) defined various methods for defining this including CRC (Class-Responsibility-Collaboration) cards for specifying objects (classes). Rather than using UML class diagrams, classes were defined on 4x6 index cards. The idea behind the cards was to simplify the design and create well defined classes that did one thing well as the specification had to fit on a single card. This change in design led to software and systems organized around cooperating networks instead of strict hierarchies.

Fast and ubiquitous communication and smaller and more powerful processes enabled the advent of the Internet of Things (IoT). This brought about a change in the configuration of the systems elements from a hierarchical to a network design. Rather than the strict hierarchical design, autonomous systems could be linked together to provide functionality in new ways. Reduced computing costs and faster networks reduced the need for closely coupled physical systems since communication delays were reduced considerably. Consequently, different architectures could be explored while still maintaining system security and reliability. Because of the nature of some systems, they will still need to be closely coupled, but the constraint has been considerably reduced. It also means that the network pattern can be used in multiple settings and applications.

## The Example Model

Powerhouse Engines (PE Inc.) is an automotive supply company providing internal combustion engines. PE Inc. finds that it has gradually become less competitive over the years largely due to their outdated technology and largely manual processes. Foreign and domestic competitors have started to cut into their business and the stakeholders are concerned that the company's loss of market share will accelerate and that they will eventually become insolvent. To combat this, the shareholders have proposed an investigation

into strategies and technologies such as, augmented reality, robotic assembly systems, 5G, artificial intelligence, data analytics, additive manufacturing, flexible manufacturing, outsourcing of select manufacturing and IT systems, Data analytics, Hybrid/electric engines, Etc. Note that this example has been used in previous papers: (Brooks, Hause, 2023), (Hause, Kihlström, 2022), (Hause, Kihlström, 2023). However, the work presented here is all new in spite of using the same example model.

Most if not all this new technology will require software as a major part of the implementation. Current manual processes will be automated, exchange of paper documents will be replaced with electronic exchanges, and telemetry along with data analytics will be deployed to help ensure that implemented changes provide faster, better, and cheaper outcomes rather than just the appearance of improvement. To help understand the impact of these changes, an architecture model will be developed documenting the existing system and where technology insertion will take place in two phases. The As-Is model has already been captured defining the existing systems and processes. The model below defines To-Be architecture.

## *Defining Capabilities*

A set of capabilities for Powerhouse Engines has been developed as shown in Figure 2. A capability is the ability to achieve a desired effect realized through a combination of ways and means (e.g., systems and other elements in the resources view) along with specified measures. Capabilities define what is desired without defining how.

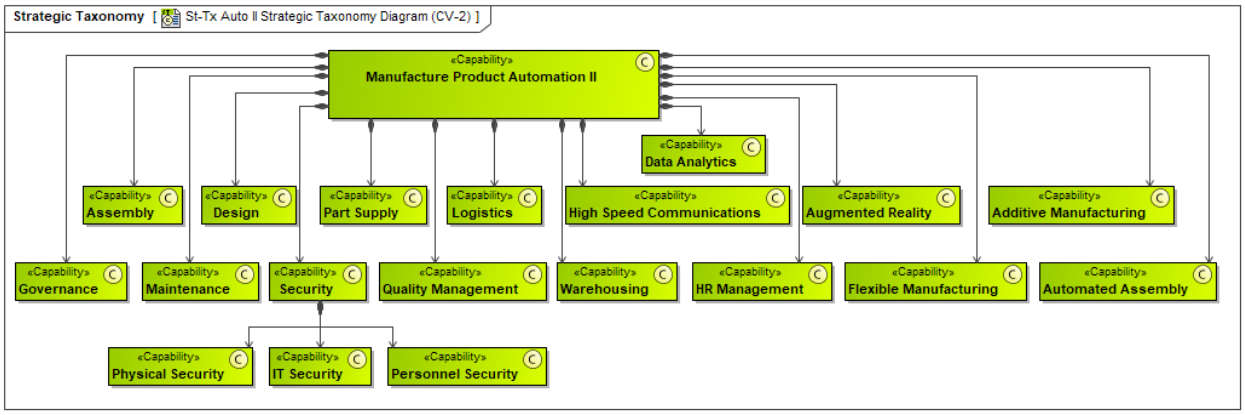


Figure 2. Factory Capabilities

The main capability for the to-be or automation phase II enterprise shown in Figure 2 is to manufacture products. This diagram reflects the final phase (Automation II) where additional capabilities described above have been deployed. The part supply capability will be implemented by a variety of different resources including external suppliers, internal casting, 3D printers, etc. Other capabilities such as assembly, design, logistics, etc. are also shown. The assembly capability can be implemented by a combination of people, tools, automation, robots, etc. These capabilities will be valid throughout the different temporal phases of the enterprise. These capabilities are mapped to operational activities which are business processes which provide value.

All elements in the operational view are solution independent. For example, the Produce Part operational activity could be provided by any of the three suppliers listed previously. The vertical boxes are swimlanes and represent operational performers. These are created based on the logical, causal, temporal, physical and other groupings of the defined activities. The operational activities define what needs to be done and the operational performers execute those activities. This performs an explicit allocation of the activities to the operational performers. The interactions shown on the activity diagram are linked to the interactions on the

structural diagram (not shown). This ensures that the proposed functional interactions can be implemented using the structural components.

These operational activities are then grouped in a logical order to define the manufacture process as shown in Figure 3. The products are procured based on a design and then are produced for assembly. The order of manufacturing documents what is reflected in the current processes and will directly relate to the assembly process no matter how it is implemented. This will form the basis of the software control logic and consequently the behavior and structure of the implementing software.

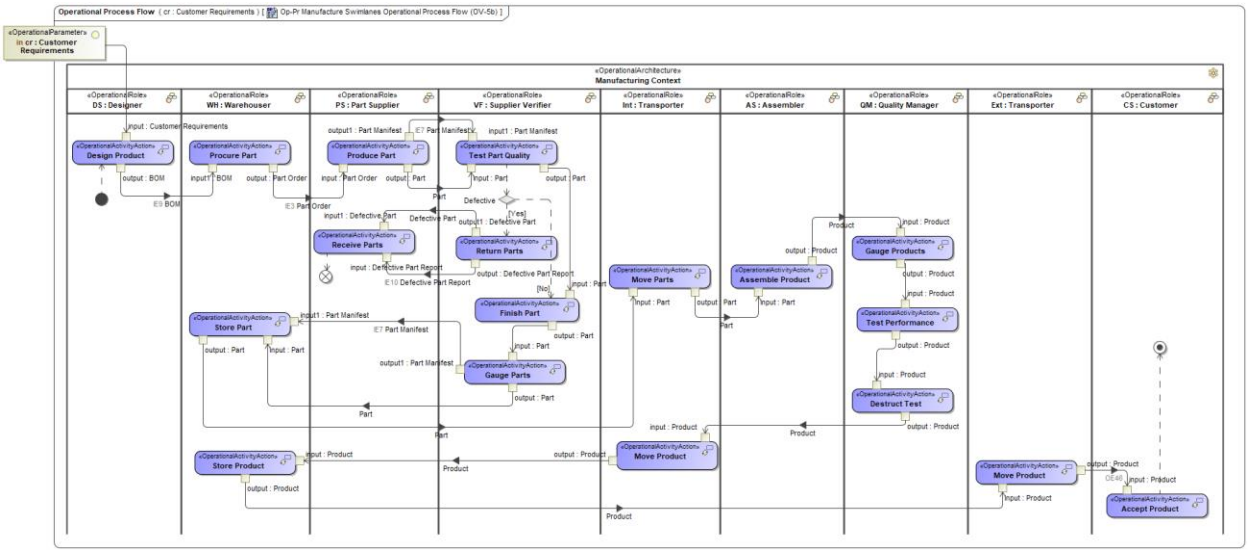


Figure 3. Engine Factory Assembly Sequence

Having used the operational views to define what must be done, the resource views are now used to define how it will be done. This is done by mapping implementation elements to the capabilities as shown in Figure 4

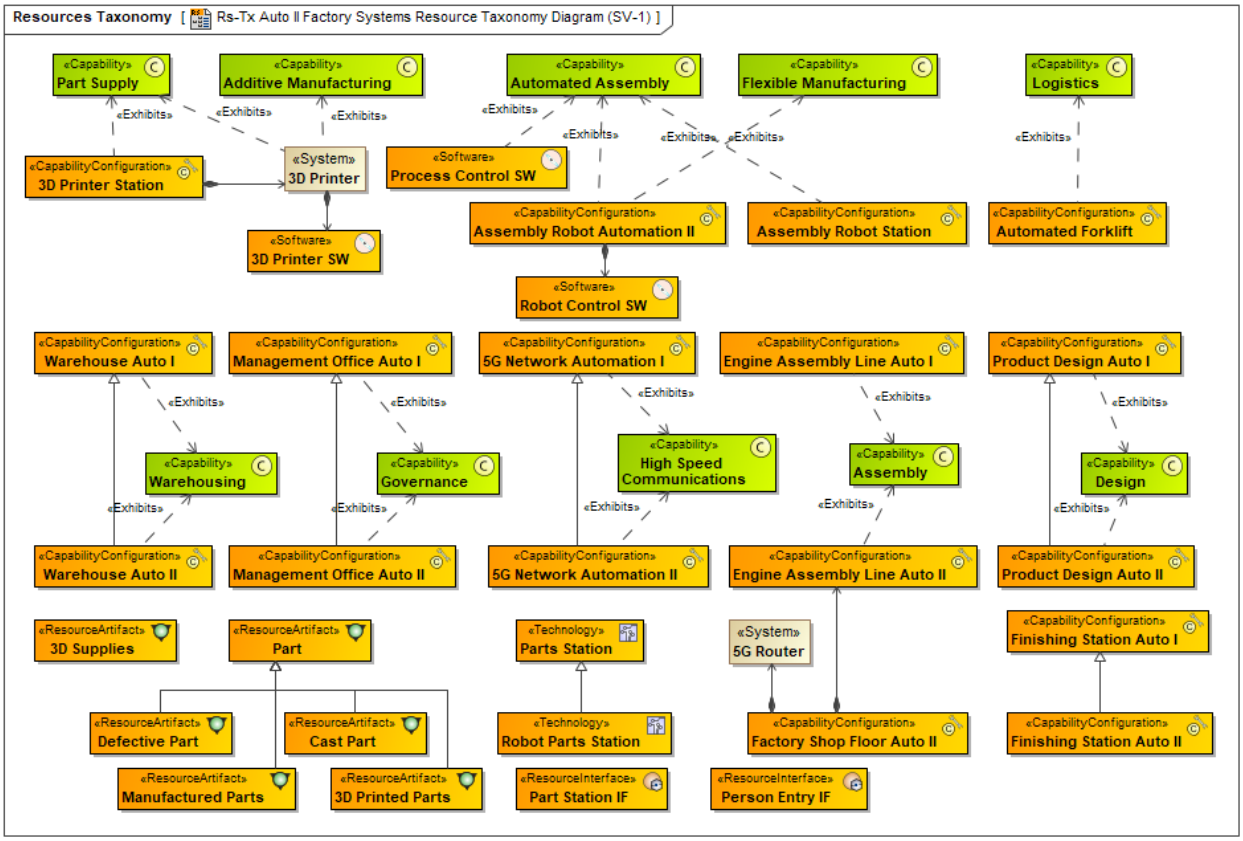


Figure 4. Mapping the Capabilities to the Implementing Resources

Figure 4 shows the resources that will implement the capabilities for the second and third phases. These are a combination of people, complex and simple systems, and technology. Some software has been defined and is shown as being a part of the systems. The architecture of the assembly plant is defined making use of these systems as shown in Figure 5.

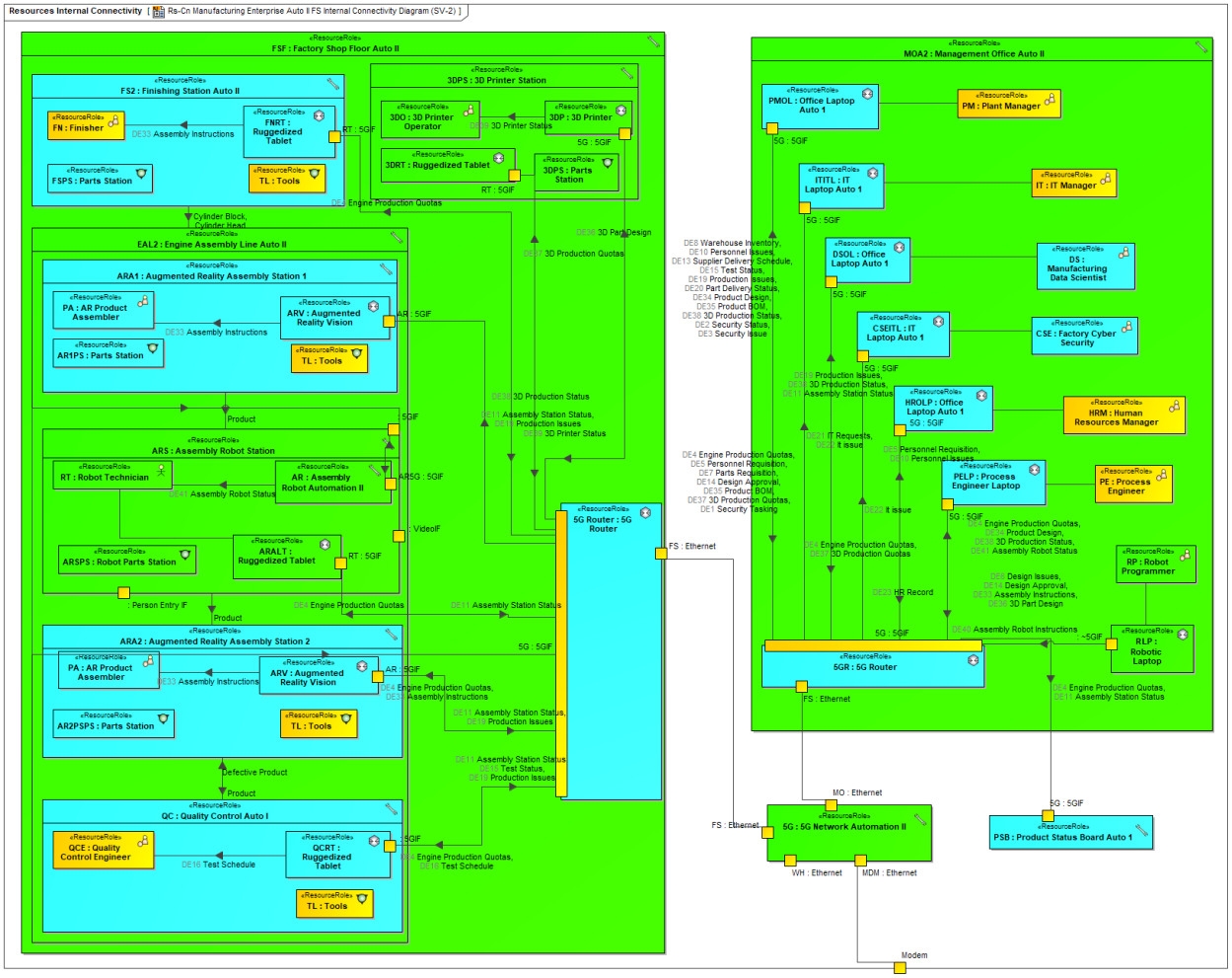


Figure 5. A Portion of the Resulting System Architecture

Figure 5 only shows a portion of the assembly plant, namely the Engine Assembly line and the Management office. The systems shown are from the final phase (Automation II). Data is defined for their interactions (Figure 10) as well as physical elements. For example, 3D Printed Parts have been defined as they will be produced by the 3D Printer and used in the assembly process. Functionality provided by the systems is defined by resource functions. These are solution specific rather than solution independent as they show how a particular solution set will implement the operational activity. What has not been defined is the deployment of software and their interactions. To do this we need to look at the software in isolation.

## *Modeling Software in the UAF*

The definition of software architecture in UAF is similar to the system architecture. It will revolve around centers of functionality grouped together to perform specific functions. For modern systems, the majority of these functions or behavior will be implemented in software. During the resource modeling stage, some software elements have been defined throughout the architecture and linked to the capabilities. However, to fully define the architecture we need to make sure that software behavior supports the capabilities. The capabilities are added to a diagram along with the existing software in the model linked to the capabilities. Missing software is then added to the model to provide the missing functionality.

In SysML, the Block can model any structural concept. UAF extends the Block to define systems, technology, and software elements. The process is similar to that which we defined above for the system architecture. Software elements are linked to capabilities to demonstrate an exhibits relationship. The required structure, behavior, interactions and relationships are defined. Sequences and constraints are defined to ensure a complete solution. For state-based elements, the behavior is defined using a state machine. This process for defining the architecture will be incremental and iterative as the process progresses. Sequence diagrams can act as a validation exercise to ensure sufficient operations, interactions, data, and software elements have been defined to implement the required functionality. Figure 6 shows the first step in the sequence which is to define the missing software elements for the capabilities.

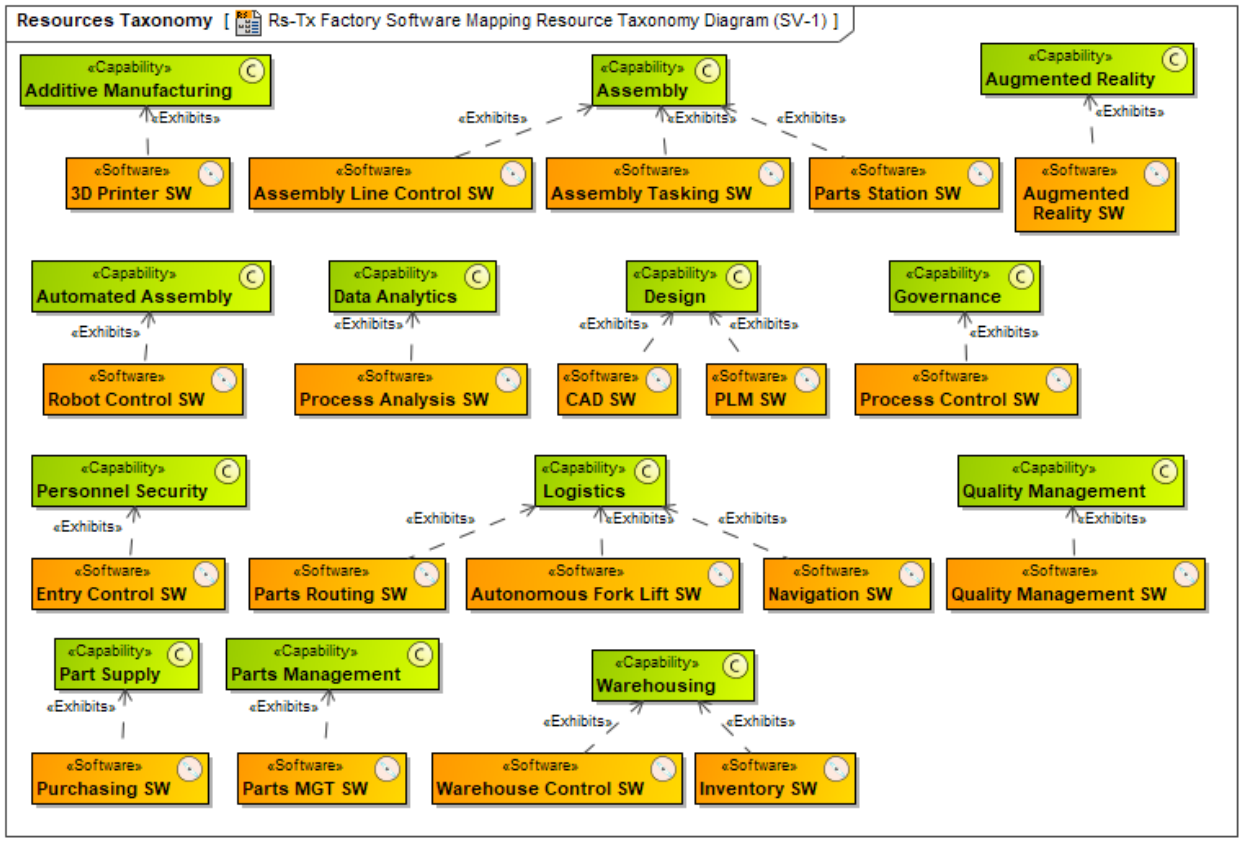


Figure 6. Mapping the Capabilities to the Implementing Software

As mentioned earlier, a well-defined class has a clear purpose and does one thing well. As part of its definition, operations, data, and associations are added to each of the Software elements as shown in Figure 7. The Resource Methods (Operations) are created to support the purpose of the software. These are used in state and sequence diagrams along with events and resource exchanges. These define major functional aspects of the software. This method is used in the Object-Oriented Systems Engineering Methodology (OOSEM) and is best-practice OO modeling. (OOSEM, 2024) This is an alternative to using activities and allocation defined earlier in the paper. This imposes a tight coupling between the structure and behavior as the resource method is owned directly by the software rather than allocated to it.

It is important to realize that the software objects created are not individual software classes but instead can represent complete software packages or application that may comprise thousands of lines of code. They simply encapsulate the required functionality without constraining the eventual implementation regardless of whether it is built or purchased. Each represents a piece of the aforementioned system intelligence.

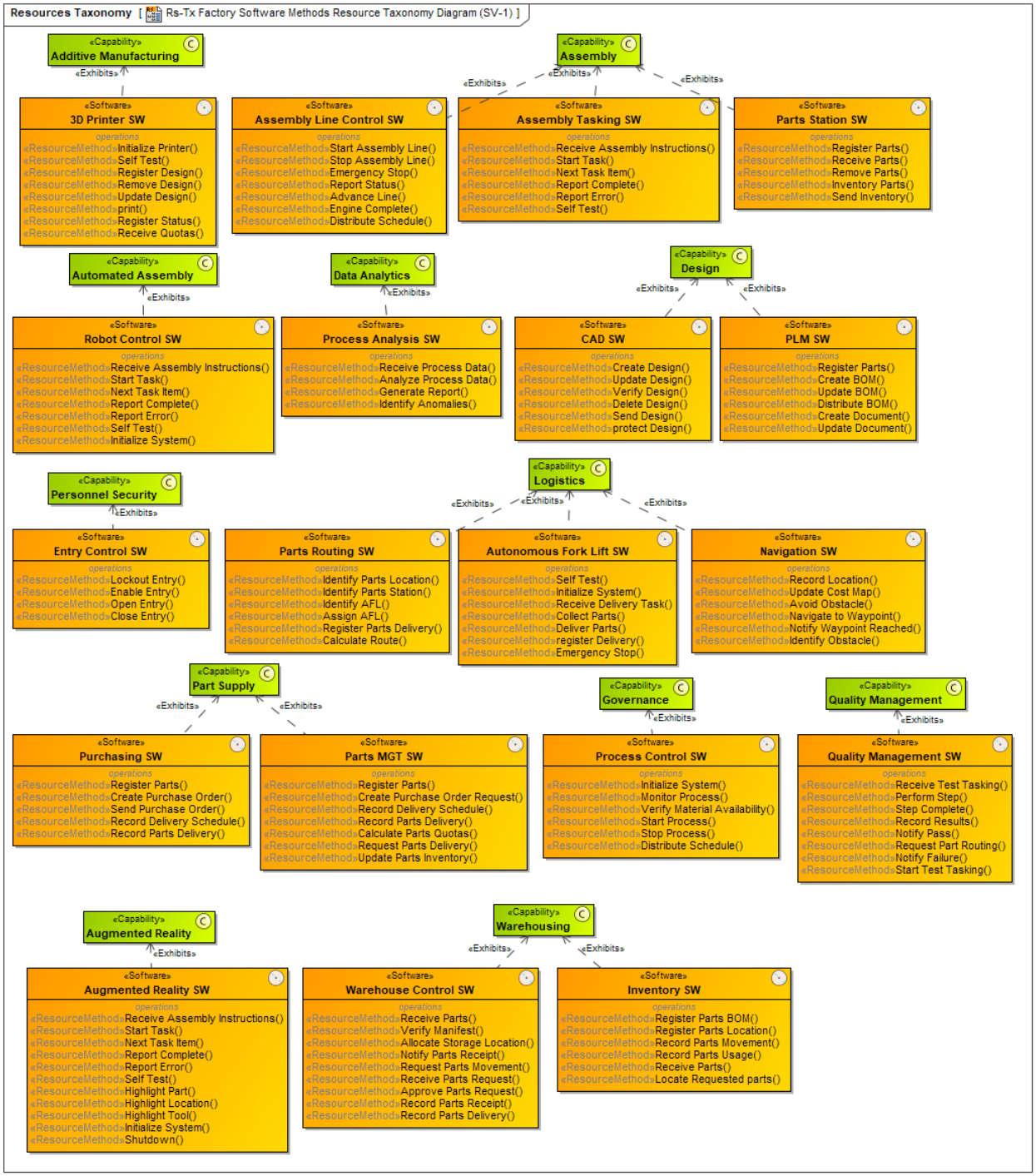


Figure 7. Elaborating the Software Elements with Operations

In Figure 8 a software architecture has been created to show the software context. This is an artificial construct as software will be distributed on processors throughout the architecture. Allocation to the systems is dependent on best practice architecture modeling.

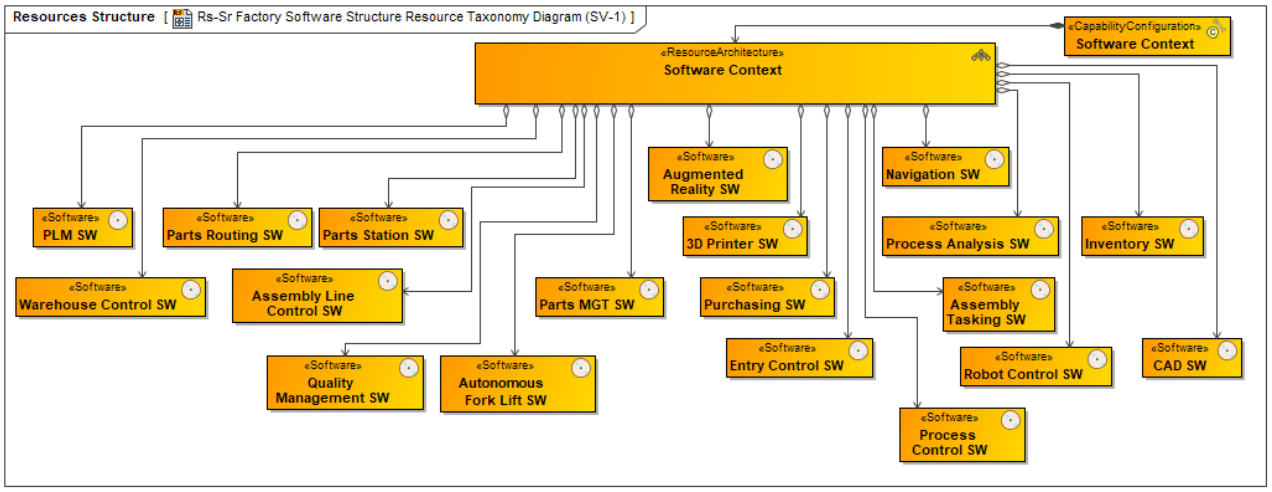


Figure 8. Aggregating the Software into a Virtual Architecture

The internal connectivity diagram is then created and populated with the defined software objects as shown in Figure 9.

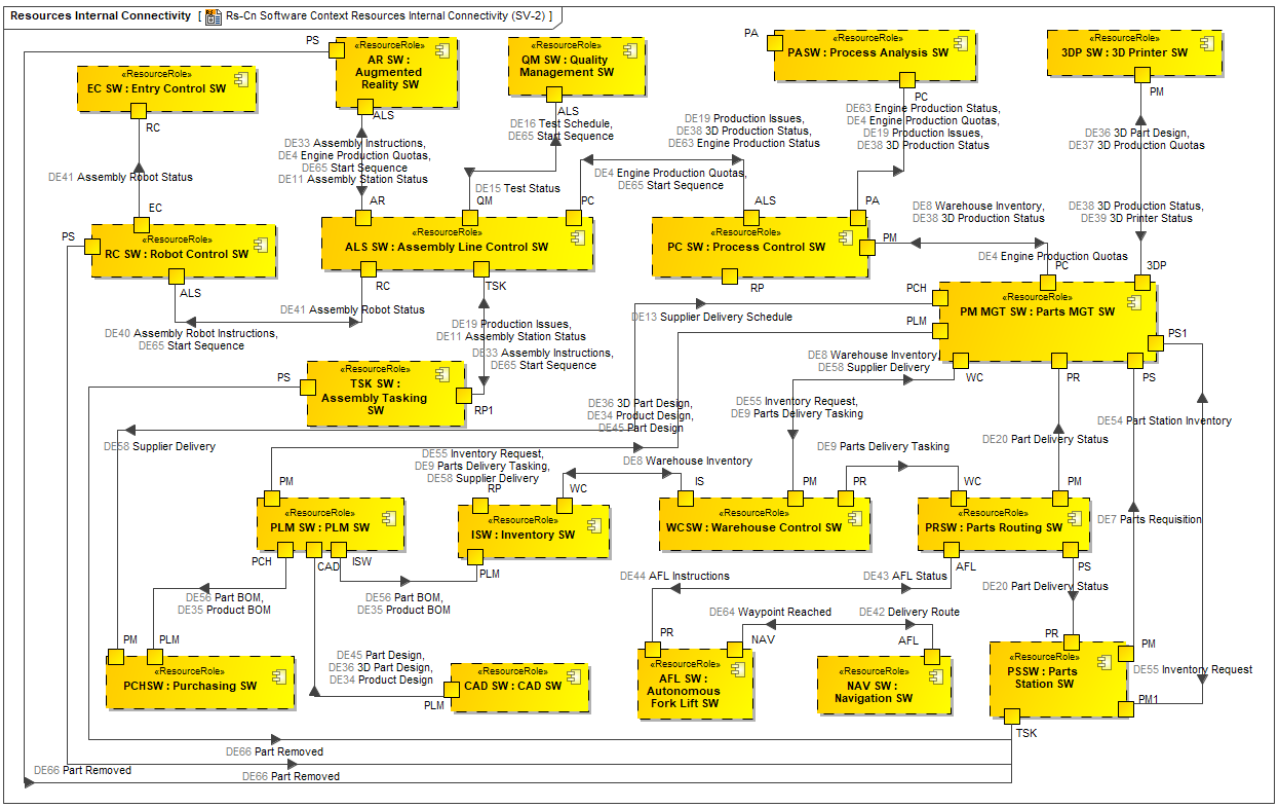


Figure 9. The Resulting Virtual Software Architecture

For software to execute, it must be deployed on a physical system. Prior to that of course, the software and system constraints must be captured as well as interactions and limitations. This diagram shows a virtual software architecture with abstract connections between software. Data interfaces are defined corresponding to the exchanged data. As the decisions regarding deployment have not been defined, protocols and

transport definitions are not relevant. The exchanged data is then added. Defining the required interactions before hardware implementation helps define required interfaces for the systems.

One of the major constraints on the system is the physical location of the machinery and the assembly process. The high-speed nature and potential safety hazards in a manufacturing environment will also force systems and controlling software to be collocated. Other non-real-time software such as CAD and PLM can be deployed in the cloud if desired. Defining these software interactions ahead of times helps create an understanding of the resulting issues that will take place between co-located software and software deployed on different systems or even in different time zones.

During the creation of the diagram and the resource exchanges, missing data element will come to light. The nature of hardware and software is quite different in that the attributes and other data relationships are required in order to have a fully defined system. This the advantage of developing these systems iteratively and incrementally. Missing elements become obvious during development. The data models are shown in Figure 10 and Figure 11.

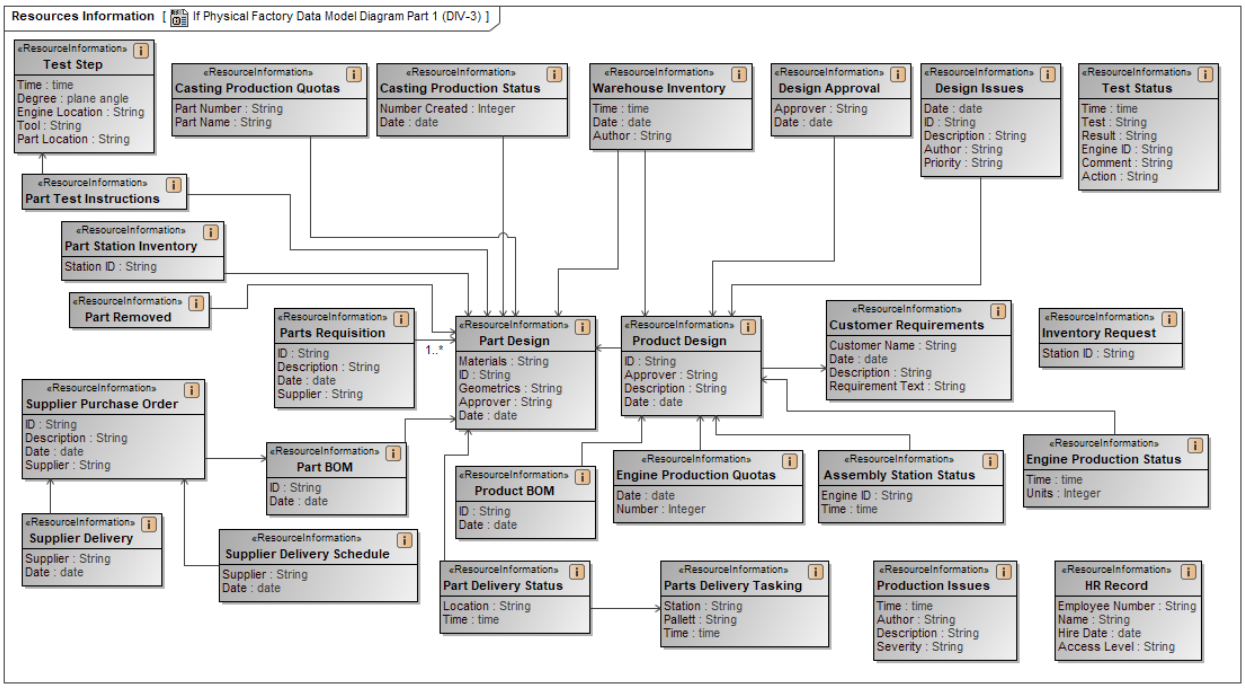


Figure 10. The Enterprise Data Model Part 1

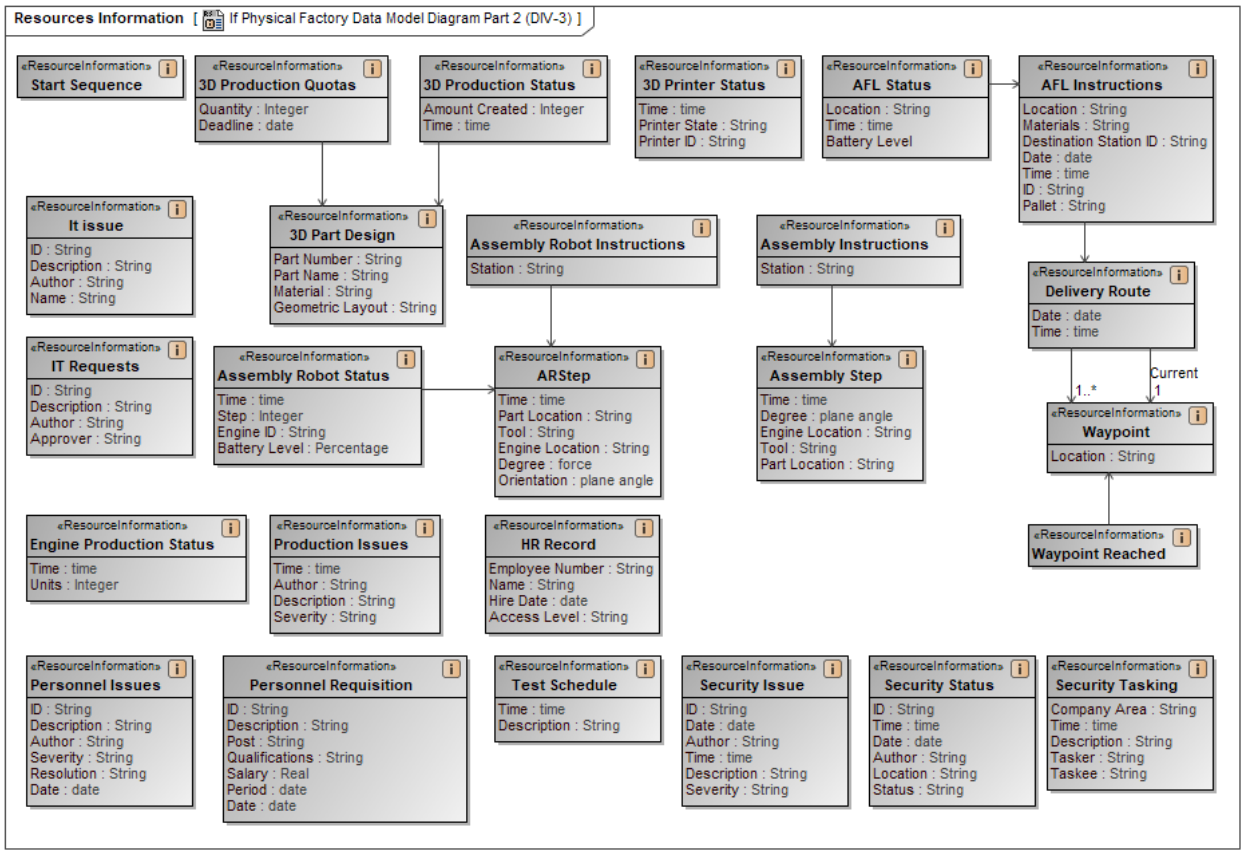


Figure 11. The Enterprise Data Model Part 2

## *Defining Interactions and Causal Sequences*

As discussed previously, static diagrams such as the taxonomy and internal connectivity diagrams define structure and interactions but do not show any order or why the interactions take place. Sequence diagrams are used to show the order in which the interactions defined on the Internal connectivity diagram shown on Figure 9 take place and when the operations defined for each software element shown on Figure 7 are executed. Multiple sequence diagrams can be created showing both sunny day scenarios as well as error sequences. Each step in the sequence can also be examined to determine error conditions and resolutions. They can also be arranged in a hierarchy to limit the complexity of the diagrams and provide reuse of the sequences. The high-level execution of the factory is shown in Figure 12.

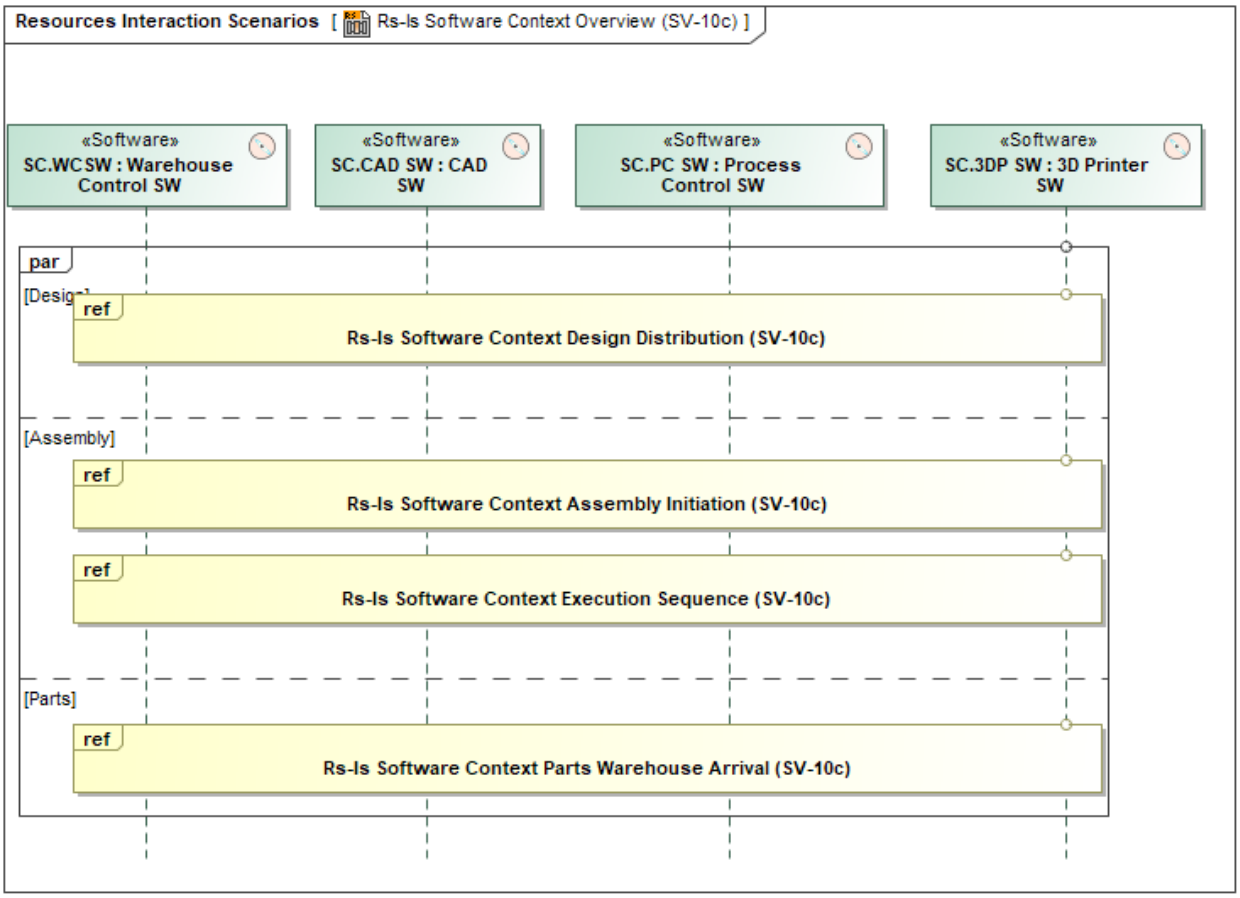


Figure 12. The Resulting System Execution Sequence

The sequence diagram uses the interaction frames to define the high-level architecture of the system functional sequences. The order will correspond to the process flow diagram shown in Figure 3 describing the factory assembly sequence. There are multiple sequences taking place at once shown in the “par” frames. The engines are being designed and the designs distributed, parts are arriving at the warehouse and being stored, and the factory is assembling engines. These are all independent sequences. At this level the details are hidden as a fully defined diagram would not be legible or in fact useful. The Design Distribution sequence is shown in Figure 13.

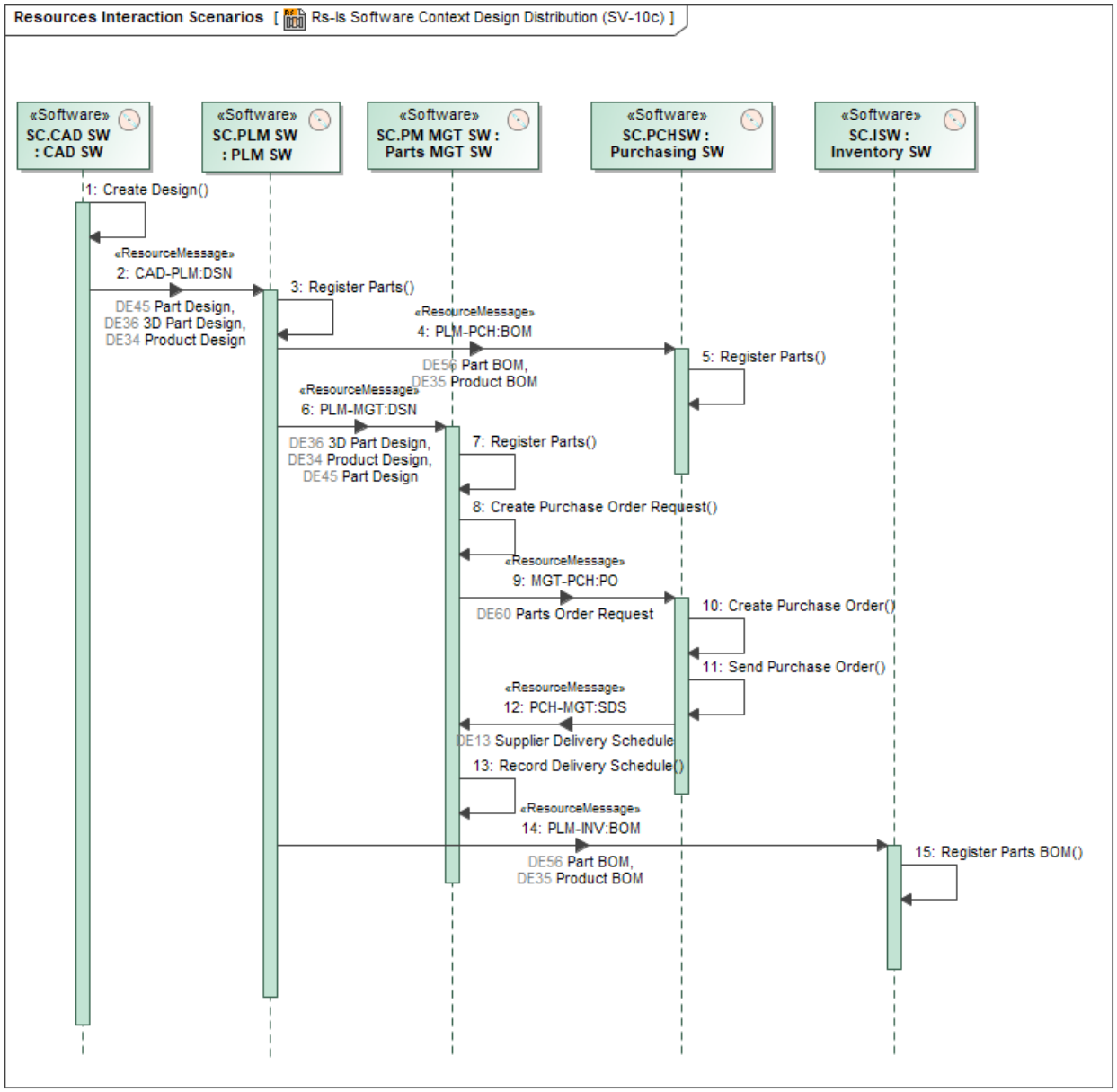


Figure 13. Design and Parts Distribution Sequence

The initiating element is the creation of the design. In general, the sequences involve operations completing and sending a message, or a message being received and executing an operation. In this case, the design is distributed to the PLM system and from there the Bill of Materials sent to parts management, purchasing and inventory to register the required parts. Registering the parts allows the system to store, order, move, etc. the different parts for an engine assembly. Demand for parts each day will correspond to the number of engines being assembled. Figure 14 shows the assembly initiation. It is important to note that the detailed execution of the factory requires multiple sequence diagrams. The model of the factory itself contains over 400 diagrams. Consequently, this is not a complete representation of the model.

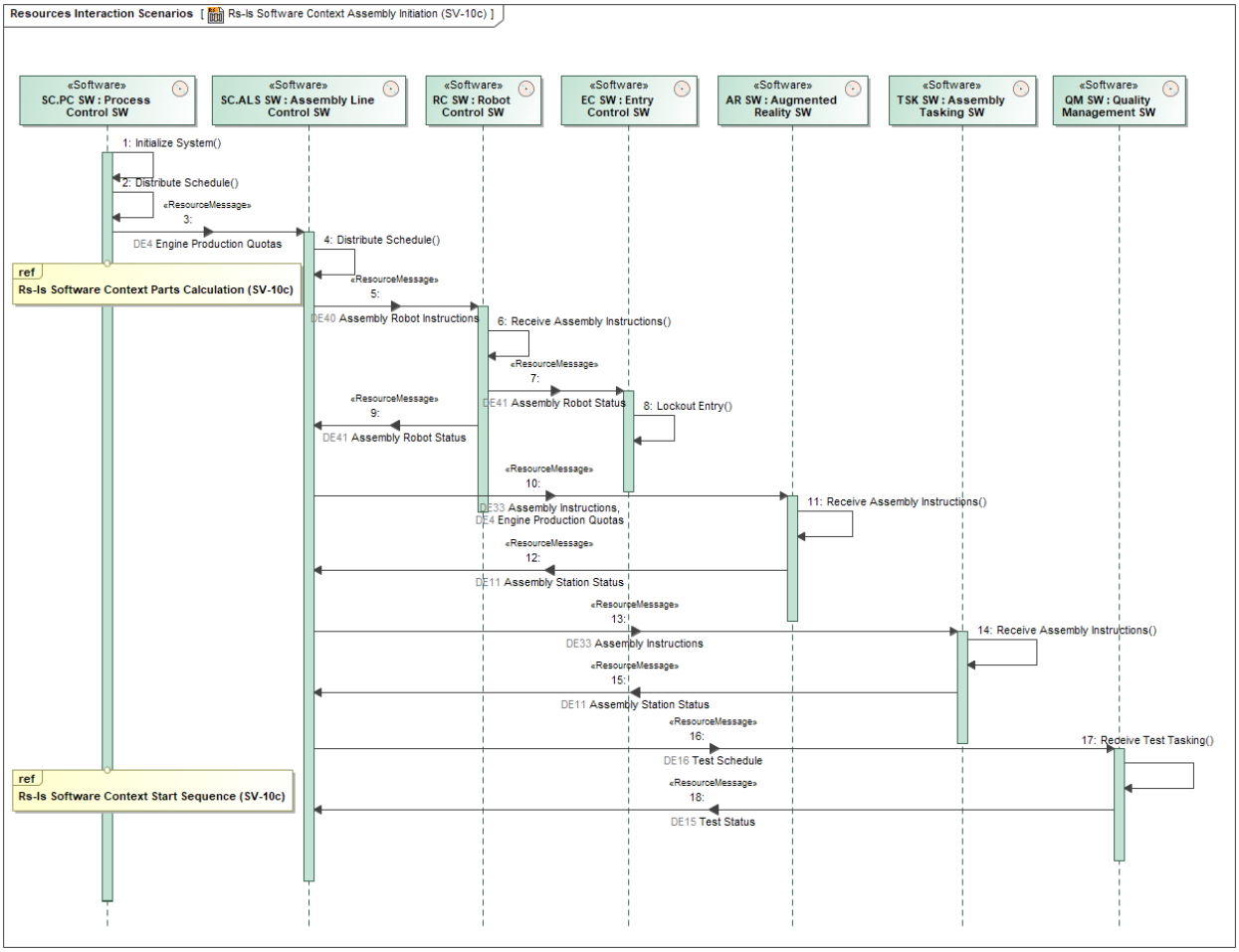


Figure 14. Engine Assembly Initialization Sequence

The process control object initiates the start sequence. This is sent to assembly line control who forwards this to all the assembly and quality stations. who respond with a status message. The aggregate status is forwarded to process control. The status will be processed, and the decision will be made whether or not to execute the factory start sequence. This is shown in Figure 15. As part of the initial set of actions, the engine quotas are received, and the required parts calculated. Other sequences will distribute the parts to the different assembly stations and if necessary, order parts from the warehouse or from external suppliers.

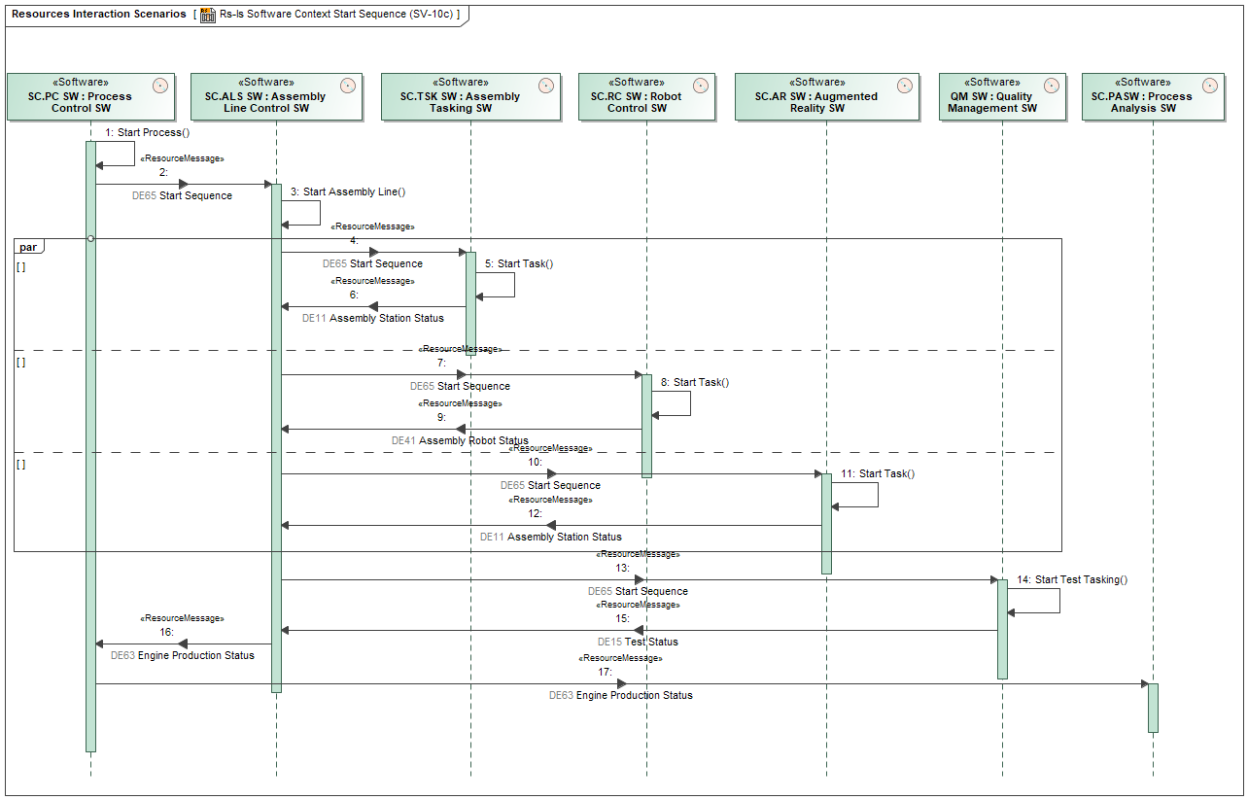


Figure 15. Factory Start Sequence

Once again, process control initiates the start sequence. This is sent to assembly line control who forwards this to all the assembly and quality stations. These respond with a status message and the aggregate status is forwarded to process control. If all is well, the assembly sequence will be executed as shown in Figure 16.

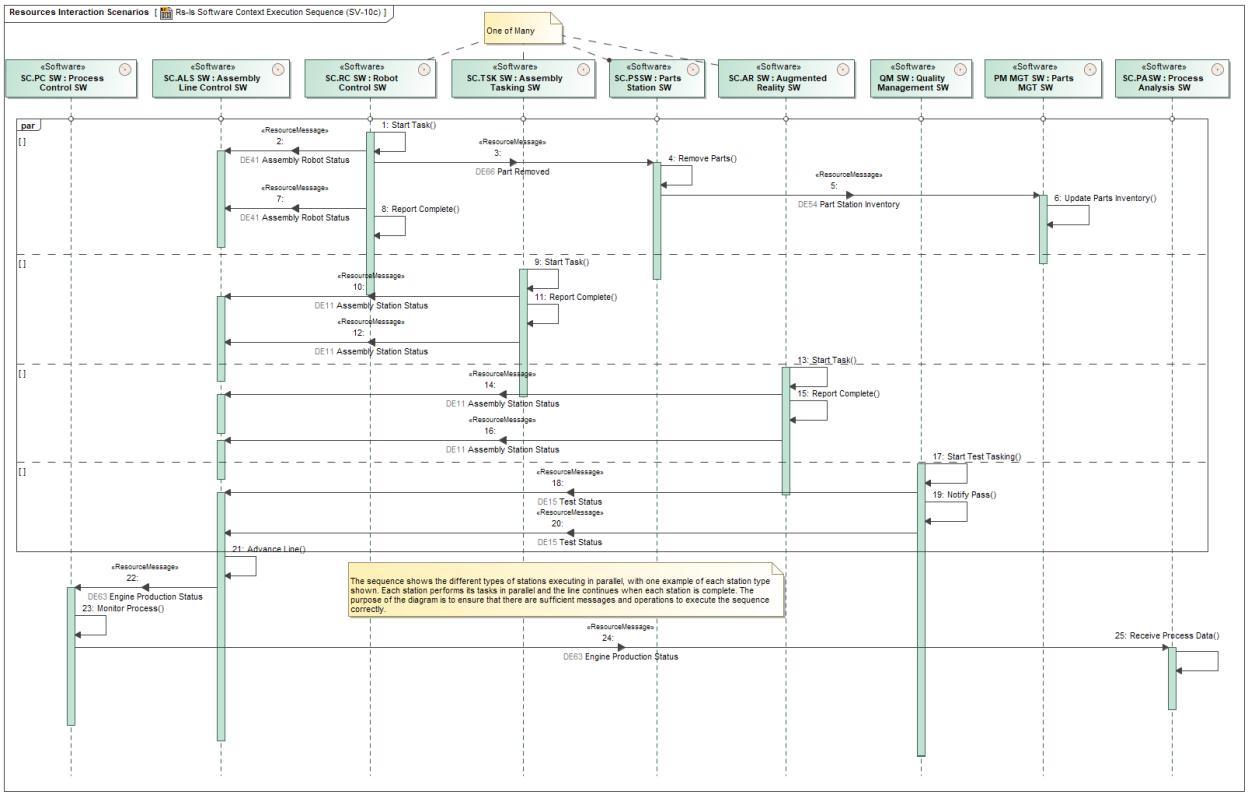


Figure 16. The Factory Execution Sequence

For simplicity, only one of each station type is shown. The different stations operate in parallel performing their tasks, with handoffs between them as the engine is assembled, parts created, and the assembled engine tested. Assembly line control is informed when each task starts and completes. When all are complete the line advances. Finally, the process information is sent to data analytics to process for quality and statistical analysis. Creating these diagrams ensures that the necessary data, software elements, operations, functionality, and interactions have been defined for the system. When data, operations, interfaces, or interactions are found to be missing, these are added to the model in the appropriate place.

## *State-Based Modeling*

State machines are used to model the life cycle of the structural elements. This of course can include software as well as systems. States correspond to collections of behavior and responses to events. State machines are not normally created for all structural entities. It is best to concentrate on those which require their behavior to be documented or if they are a major component. Not all states and errors are shown in this diagram. The purpose of the diagram is to understand the main phases of the assembly sequence and the behavior that takes place there. State machines can be executed in most modeling tools to validate the required behavior. If desired, the various state machines can be connected and executed together to determine how the various components work together. The state machine for the augmented reality assembly software is shown in Figure 17.

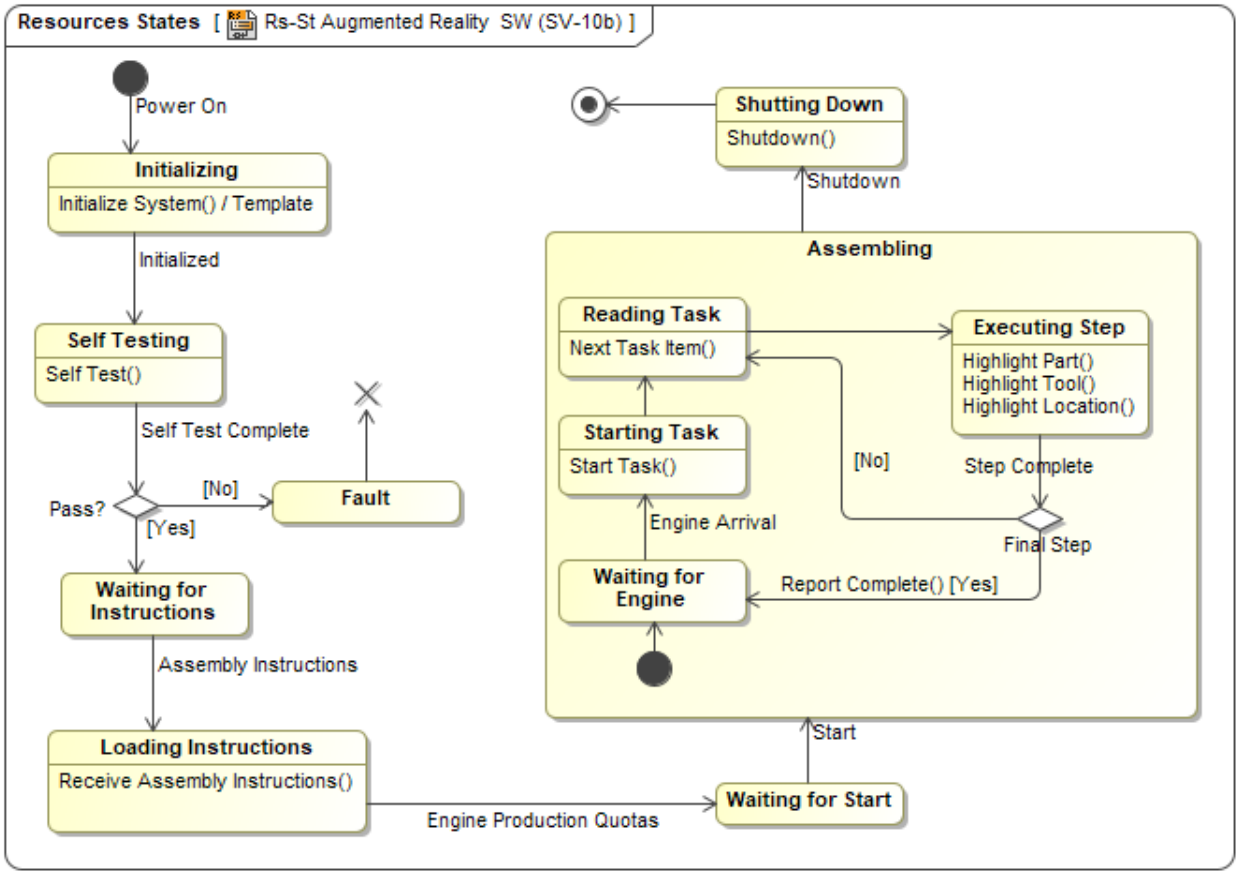


Figure 17. Augmented Reality Assembly State Diagram

## *The Use of Service Oriented Architecture in Software Specification*

The concept of a service is used in a variety of different ways and has different meaning to different persons. An attempt to get different people to provide a definition for the word service usually ends up with as many definitions as there are persons asked to provide it. Many will state that its purpose is to define internet software services. However, it does this and so much more. The concept of service-oriented architecture (SOA) does however represent a restriction as to the meaning of services within the context of SOA as it is implements in the UAF. (Hause, Kihlström, 2021).

A service within the context of SOA has a set of agreed upon properties:

- It deals with a repeatable business activity with specified outcomes, usually either a successful one or an unsuccessful one.
- It is self-contained, i.e., all the functionality is within it.
- It is a black box as far as consumers are concerned. This means that the consumer has no knowledge of the inner workings of the service.
- It is loosely coupled i.e.; it can be treated as a single separable element.
- It may be composed of other services, but this is invisible to external consumers.

A service is associated with a service contract that can deal both with functional and non-functional parts. The functional part from the point of view of the consumer is the interface that exposes the service to the consumer, the non-functional parts deals with performance issues that the service offers. The service

concept can be used both within a logical architecture as well as within a physical architecture. Within the logical architecture the service is used from a pure business logic perspective whereas in the physical architecture both business logic and the realization of the service is being described. A realization of a service can be made either as a separate service that can be accessed through a defined interface. A more advanced service implies the use of a service broker that based on the consumers indicated needs will create a service from a set of services under its control that will create the outcome the consumer desires.

A related concept is named micro-services. Micro-services provide an architectural style that structures an application as a collection of services that are independently deployable, loosely coupled, organized around business capabilities, and owned by a small team. This means that is essentially about breaking down large applications into several smaller ones. The fact that they are independently deployable implies that each service can be update and reorganized independently from the other ones. Making use of micro-services efficiently requires the creation of a well-defined communications infrastructure and an appropriate partitioning of the total application. It also requires that each micro-service owns its own internal data completely and that data to be published has an agreed format that can be interpreted by all micro-services that needs to use it. Publication is generally based on asynchronous handling but there are patterns as well for synchronous publications. For further information see Hause, Kihlström (2022).

Looking at the applications defined for the factory in Figure 9, the upper half largely relates to applications with real-time requirements. The bottom half of the figure discusses things like entry control, parts routing, purchasing software, parts management, warehouse control and inventory. Parts of these can be service based and since the applications are quite large might be broken up into micro-services. Another possibility here is to transfer some of these applications to a cloud, i.e., make access to these applications reliant upon the internet with the hardware deployed elsewhere. The architecture for a cloud consists of a front-end platform where clients can access the cloud via the Internet is a secure fashion, and a back-end platform that contains servers as well as storage for data.

There are several ways that the cloud can operate. The software for the applications can be placed directly in the cloud eliminating the need for local servers and storage. The software is no longer deployed locally, and the access is provided via a web interface. Another way is to use the cloud as a platform with a given set of solutions and middleware allowing the users to deploy their application software on top of the platform making use of the facilities that the platform provides. This means that the end user is responsible for deployment and configuration. There are also clouds that only offer basic infrastructure and let the end user needs manage the applications completely within this infrastructure. Parts of the applications for the factory example could be dealt with by either of these possibilities as cloud-based applications.

## *Remaining Work*

The model is not yet complete and there are a number of tasks still remaining. These include:

- Create cloud-based and other deployments.
- Create additional state diagrams for software elements.
- Create additional sequence diagrams for other scenarios.
- Create additional signals for messages causing state changes.
- Create and populate resource interfaces based on I/O from architecture diagrams.
- Create a simulation of sequence, state, and structure diagrams.
- Allocate software to HW elements.
- Add software messages to existing HW structure diagrams.
- Iterate and improve the model over time.

## Summary and Conclusions

This paper examined some of the issues concerning the modeling of software in an enterprise setting. It showed how the initial solution-independent model created in the operation views can be used to describe the overall required behavior. We then examined how the capabilities are the main driving force behind the model with the various aspects relating back to it. It then showed how a software architecture can be created using these techniques, and how sequence diagrams can be used to validate the defined software architecture to ensure that it provides the required functionality. Finally, it looked at how service-oriented architectures can be used to encapsulate defined software functionality for deployment in the cloud or via other architectures.

The definition of software in models was previously best practice and ubiquitous. With the advent of Agile programming software models becoming less and less common. This has resulted in software architecture that are either documented in the code or in the heads of the designers. This means that when people leave as they all eventually do, this knowledge can leave with them costing time and money when others must figure out how the software works. Defining a software architecture need not be ponderous or a mammoth undertaking. As with all things, it is knowing the level of detail required and what is actually useful to model. Finally, bear in mind that the purpose of the project is to rearchitect the business, the factory, the supporting infrastructure, the people, and finally the software. The purpose of the model is to de-risk the exercise by modeling the systems, finding errors, examining alternatives, and rearchitecting again. This requires several passes, but doing this in the model is far easier and less costly than rearchitecting the physical factory.

## References


Brooks M., Hause M., 2023, Model-Based Cyber Security at the Enterprise and Systems Level, presented at the INCOSE International Symposium, Honolulu, Hawaii

Campbell G., 2004, Carnegie Mellon University, Reconsidering the Role of Systems Engineering in DoD Software Problems, online, available from https://insights.sei.cmu.edu/documents/2931/2004_017_001_22631.pdf

Charette R., 2023, IEEE Spectrum, How Software is Eating the Car, available online at https://spectrum.ieee.org/software-eating-car

Conant L., 2023, Carparts.com, When Were Vehicles First Equipped with Computers?, Available online from https://www.carparts.com/blog/when-were-vehicles-first-equipped-with-computers/

Conway M., 1968, How Do Committees Invent?, Originally published in Datamation Magazine, Available online from https://www.melconway.com/Home/pdf/committees.pdf

Eloranta V., Koskinen J., Leppänen M., Reijonen V. (2014). Designing distributed control systems: a pattern language approach. Wiley series in software design patterns. Chichester: Wiley. ISBN 978-1-118-69415-2.

Fowler M., Scott K., 1997, UML Distilled, Applying the Standard Object Modeling Language, Published by Addison-Wesley Corporate and Professional

Friedenthal, S., Moore, A., Steiner, R. Practical Guide to SysML: The Systems Modeling Language Second Edition, Morgan Kaufman, Oct 31, 2011

Hause M., Thom F., 2008, Building Bridges Between Systems and Software with SysML and UML, presented at the 2008 INCOSE Symposium in the Netherlands.

Hause, M. 2014. “SOS for SoS: A New Paradigm for System of Systems Modeling.” Paper presented at the IEEE, AIAA Aerospace Conference, Big Sky, US-MT, 1-8 March.

Hause, M., F. Dandashi, 2015. “UAF for System of Systems Modeling , Systems Conference (SysCon).” Paper Presented at the 9th Annual IEEE Systems Conference, Vancouver, CA-BC, 13-16 April.

Hause, M., Kihlström, L., 2021, An Elaboration of Service Views within the UAF, presented at the 2022 INCOSE International Symposium, A Virtual Event.

Hause, M., Kihlström, L., 2022, You Can't Touch This! - Logical Architectures in MBSE and the UAF, presented at the 2022 INCOSE International Symposium, Detroit, Michigan, USA.
Hause M., Kihlström L., 2023, Modeling System Configurations Over Time, presented at the INCOSE International Symposium, Honolulu, Hawaii
INCOSE 2015, Systems Engineering Handbook Fourth Edition, Published by Wiley
INCOSE 2021, Systems Engineering Body of Knowledge, SEEBOK
INCOSE, 2024, Object-Oriented SE Method Working Group, Available online from https://www.incose.org/communities/working-groups-initiatives/object-oriented-se-method
ISO 15288 2023, Systems and Software Engineering — System Life Cycle Processes, International Standards Organization (ISO), ISO/IEC/IEEE FDIS 15288-2023.
ISO 42010 2022, Software, Systems and Enterprise — Architecture Description, International Standards Organization (ISO), ISO/IEC/IEEE 42010-2022.
ISO 42020 2019, Software, Systems and Enterprise — Architecture Processes, International Standards Organization (ISO), ISO/IEC/IEEE 42020-2019.
OMG 2019, *Systems Modeling Language, Version 1.6*, Object Management Group, https://www.omg.org/spec/SysML/About-SysML/.
——— 2022a, *Unified Architecture Framework, Version 1.2*, Object Management Group, https://www.omg.org/spec/UAF/About-UAF/.
——— 2022b, *Unified Architecture Framework Modeling Language, Version 1.2*, Object Management Group.
——— 2022c, *Enterprise Architecture Guide for the Unified Architecture Framework (Informative), Version 1.2*, Object Management Group.
——— 2022d, *Unified Architecture Framework Sample Problem (Informative), Version 1.2*, Object Management Group.
Stafford E., 1982, Telecontrol data Acquisition (SCADA) and Telecommunications for Oil and Gas Production, Published by EDG Engineering International Ltd.

## Biography

**Matthew Hause.** Matthew Hause is a principal consultant at SSI, a Co-Chair of the UAF group, and a member of the OMG SysML specification team. He has been developing multi-national complex systems for almost 40 years as a systems and software engineer. He worked in the power systems industry, command and control systems, process control, SCADA, military systems, and many other areas. His role at SSI includes consulting, mentoring, standards development, specification of the UAF profile and training.

**Lars-Olof Kihlström**. Lars-Olof Kihlström is a principal consultant at CAG Syntell where he has worked since 2013, primarily in the area of MBSE. He has been a core member of the UAF group within the OMG since its start as the UPDM group. He was involved in the development of NAF as well as MODAF. He has worked with modelling in a variety of domains since the middle of the 1980:ies such as telecommunications, automotive, defence as well as financial systems. He is specifically interested in models that can be used to analyze the behavior of system of systems.