Chapter 8
AADL: A Language to Specify the Architecture of Cyber-Physical Systems

Dominique Blouin and Etienne Borde

Abstract This chapter is devoted to formalisms for describing system architectures, and in particular to the Architecture Analysis and Design Language (AADL). AADL is an Architecture Description Language (ADL) well suited for the modelling of embedded and cyber-physical systems. The architecture is central in Multi-Paradigm modelling for Cyber-Physical Systems as it provides a description of the overall system and the environment into which it will operate. From such description, other models of other languages and formalisms such as those described in this book can be generated and augmented to study other aspects of the system, which is essential for its validation and verification. After a brief introduction to ADLs and their role in MPM4CPS, the AADL will be presented and its use illustrated with the modelling, analysis and code generation for a simple Lego Mindstorm robot for carrying objects in a warehouse. A simple top-down architecture-centric design process will be followed starting from the capture of stakeholder goals and system requirements followed by system design, design analysis and verification and finally automated code generation.

8.1 Learning Objectives

After reading this chapter, the reader should have sufficient knowledge of the family of AADL languages and modelling approaches for:

- Modelling a simple cyber-physical system at the different levels of abstraction of requirements, functional architecture, physical architecture, software architecture and deployment.
- Modelling reusable component libraries and product families.
- Understand basic timing and scheduling concepts and AADL properties for performing scheduling and latency analysis from the aforementioned models.
- Understand basic code generation concepts and AADL properties for performing automatic code generation with the RAMSES tool [235] from the aforementioned models.

8.2 Introduction

This chapter introduces Architecture Description Languages (ADLs) and in particular the SAE Architecture Analysis & Design Language (AADL). We start by a short introduction on the problems of building nowadays complex cyber-physical systems and explain the overall approach to solve these problems as promoted by the AADL community, which is based on architecture-centric virtual integration.

We then move on to the introduction of AADL by a short overview of the language followed by a detailed introduction of the language constructs and semantics. We have chosen to introduce AADL by presenting a

Dominique Blouin and Etienne Borde
Telecom ParisTech, Paris, France
e-mail: {dominique.blouin,etienne.borde}@telecom-paristech.fr

© The Author(s) 2020

P. Carreira et al. (eds.), *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*,
https://doi.org/10.1007/978-3-030-43946-0_8

typical development process so that it better illustrates how the language can be used. Therefore this process is first introduced followed by the Lego Mindstorm line follower robot example that is used to illustrate AADL modeling. We then present the detail modeling of the example for each step of the development process. Note that we have chosen to introduce the language constructs in an incremental way, by only introducing the constructs that are necessary for the modeling corresponding to the current step of the development process. In this manner, the reader can immediately see how the constructs are used in the context of the development process.

We illustrate the modeling process by presenting successive refinement steps of the models down to automatic code generation. Traceability links established between each step are also illustrated as well as basic analysis capabilities of AADL on the models of the deployed system. In the end, C code is automatically generated that can be compiled and deployed on the Lego robot for building a real working system.

8.2.1 Increasing Systems Complexity and Unaffordable Development Costs

The complexity of CPSs is constantly increasing due to the increasing number of functions that these systems are required to perform, which often must include more and more intelligence and must be more and more interconnected. In addition, these systems must satisfy an increasing number of constraints due to their operating environments, which are often hostile and limited in resources. Therefore, these systems are becoming more and more difficult to develop at affordable costs. This is particularly true for the avionics domain, for which a measure of complexity can be obtained from the number of Software Lines of Code (SLOC) embedded in aircrafts. This increase in complexity is illustrated in figure 8.1 where a plot of the number of onboard SLOCs as a function of years for the most common aircrafts built by Airbus and Boeing since the 70's is shown. For each constructor, the slope of the curve indicates that the number of SLOCs has roughly doubled every four years resulting in a non-linear increase in systems complexity.

The development effort required to develop these systems has been shown to increase exponentially with the number of SLOC. For example, while the F35 military aircraft has approximately 175 times the number of SLOC of the F16, it is estimated that it required 300 times the development effort of the F16 [253]. The result of this, as shown in figure 8.1, is that we are no longer able to develop more complex aircrafts with traditional development methods, since their development costs is not affordable. This is illustrated on the figure by the dark blue line whose slope is pegged after roughly year 2010.

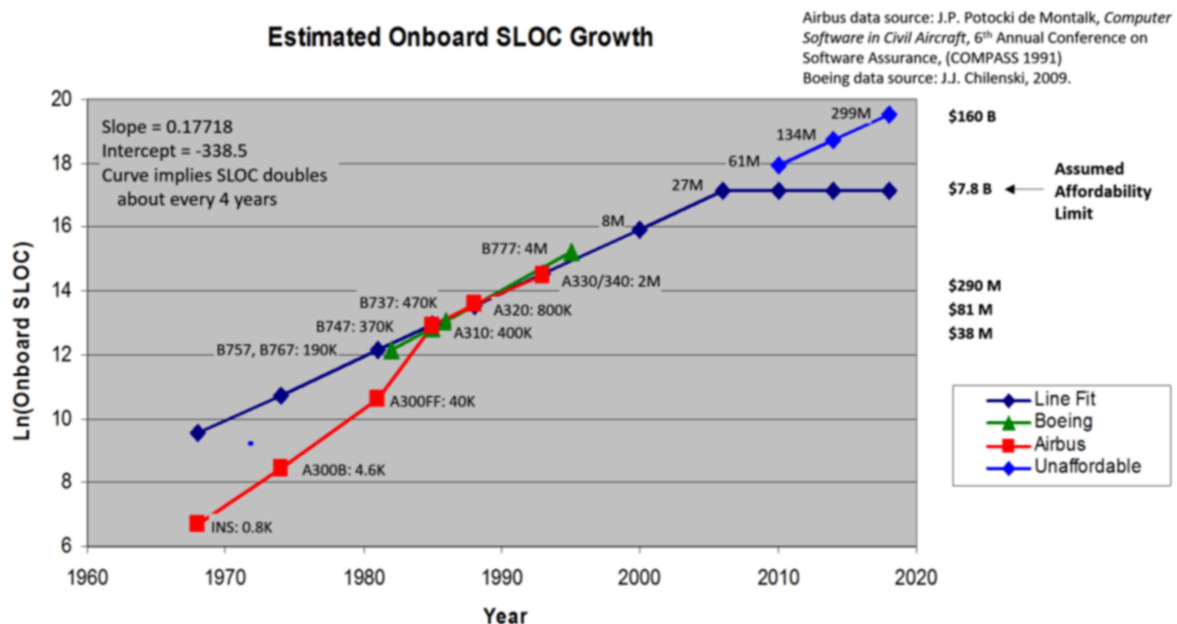


Fig. 8.1: The evolution of the number of SLOCs with time in commercial aircrafts (reproduced from [133])

In order to understand how this limit is reached, let us consider the traditional V-cycle model (figure 8.2), which is the most commonly used engineering process for safety-critical embedded systems. The numbers in blue on the figure indicate the percentage of errors introduced at the various phases of the cycle using the traditional development methods. As can be seen, a large majority of these errors (70%) are introduced at the *early* phases (requirements engineering and design) of the cycle, while the majority of these errors are only discovered much later at system integration and operation time. As a result, the cost of fixing these errors is dramatically high, since they often require the upfront design to be modified and parts of the system to be re-implemented. As a matter of fact, studies have shown for large projects that on one hand, rework due to introduced errors may account for 60% to 80% of the total software development costs [133]. On the other hand, the cost of software development, which could be as high as 70% in 2010 keeps increasing and could reach up to 88% of the total system-development cost by 2024 [133]. These figures show the potential for achieving high cost reductions by the discovery of flaws as early as possible during development.

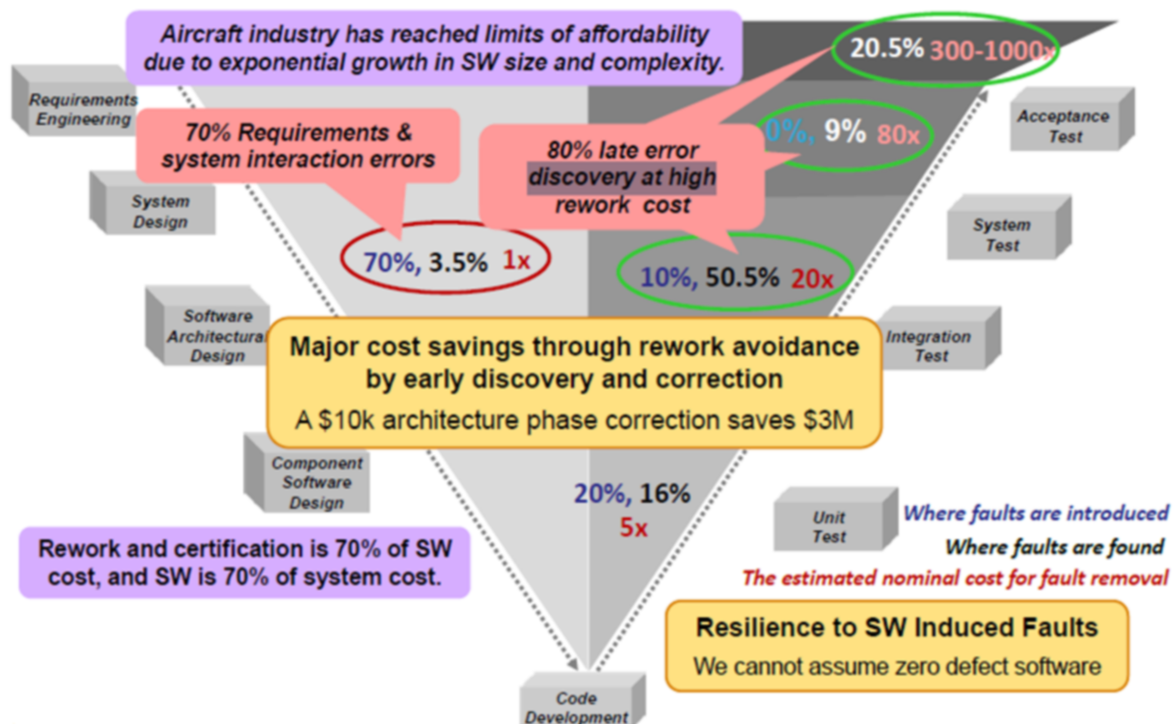


Fig. 8.2: V-cycle development process model annotated with figures for introduced errors and costs to removed them (reproduced from [101])

8.2.2 Mismatched Assumptions in Collaborative Engineering

But how are these errors leading to these rework costs are introduced? In order to understand this, let us consider a number of well-known errors that led to important damages and costs.

Ariane 5 Rocket

In June 1996, the Ariane 5 rocket exploded during its inaugural flight less than 40 seconds after departure. This was due to an integer overflow in the software used to control the side velocity of the rocket. That error propagated to an error in the navigation system, which triggered self-destruction of the rocket to prevent ground damages. This software bug cost 370 million dollars and is known to be the most expensive bug in history [10].

The integer overflow occurred because the software of Ariane 4, which had much smaller velocities, was reused for Ariane 5. It was forgotten that this software was working under the *assumption* of a maximum velocity

that could be represented in software. Such assumption was no longer matched by Ariane 5, which had much greater velocities.

iTunes

When dual core processors were first used in computers, the iTunes software was crashing randomly when ripping music CDs [156]. The software was multi-threaded with one thread determining the sound level of tracks while a second one was converting the audio data. A single core processor would always sequentially execute the first task and then the second one. On a dual core, the two concurrently executing threads were attempting to update the same music data without proper synchronization.

In this case, the software was implicitly assuming sequential execution of tasks, which assumption was no longer matched by dual core processors.

Airbus 380 Cables

When the first Airbus 380 aircraft was assembled in Toulouse in France, the wires and their harnesses turned out to be a few centimeters too short for the cabin, which led to several billions of extra costs [5].

The problem was traced to the fact that different design teams had used different versions of a Computer Aided Design (CAD) tool to create the drawings. Version 5 of the tool was a rewrite of version 4 and the calculations of bend radii for wires as they follow the air frame were inconsistent across the two versions. It was eventually realized that the issue was pervasive throughout the design. Again, this issue can be traced to different assumptions on the way calculations of wire bending are performed between different teams using different tools.

These few examples show that an important source of errors is due to assumptions a system or component makes for proper operation not being matched by the environment in which it is used. This points out the need for improved system integration performed early for ensuring consistency of the overall system is preserved during the constantly evolving collaborative designs.

8.2.3 System Architecture Virtual Integration: Integrate, Analyze then Build

A solution to this mismatch assumptions problem is promoted by System Architecture-centric Virtual Integration (SAVI), which makes use of models of the systems that can be virtually integrated and analyzed for early fault discovery before the system is physically built [253]. Such process is illustrated in figure 8.3 where the left side of the well-known V-cycle development process is augmented with parallel left side validation activities making use of models for the domains covered by the system at the appropriate levels of abstraction at each phase.

8.2.4 Architecture-Centric Authoritative Source of Truth

The aforementioned virtual integration process is supported by an architecture model playing the role of Authoritative Source of Truth (ASoT) [253]. The model provides a global view of the system and its environment and is a central place into which other models detailing the different parts of the system can be integrated. It forms a reference model capturing all properties relevant for determining if the system meets its requirements.

Such ASoT model is depicted in figure 8.4. Other models for the various analyses required by the virtual integration process may be generated from the architecture model. Properties such as safety, security, real-time performances, resources consumption, etc. estimated from the analysis activities supporting virtual integration are then used to determine if the system meets its requirements.

This architecture-centric approach therefore makes architecture models first class citizens in the development of CPSs in order to support virtual integration activities, thus reinforcing the importance of ADLs, which is the topic of this chapter. Such architecture-centric virtual integration process can actually lead to substantial system development costs reduction as shown by a study on the Return of Investment (ROI) for the SAVI Initiative [133]. According to this study, the cost reduction for a 27-MSLOC system can be as high as $2 billion out of an estimated $9 billion total cost, which represents a cost saving of about 26% due to early correction of requirements and design faults.

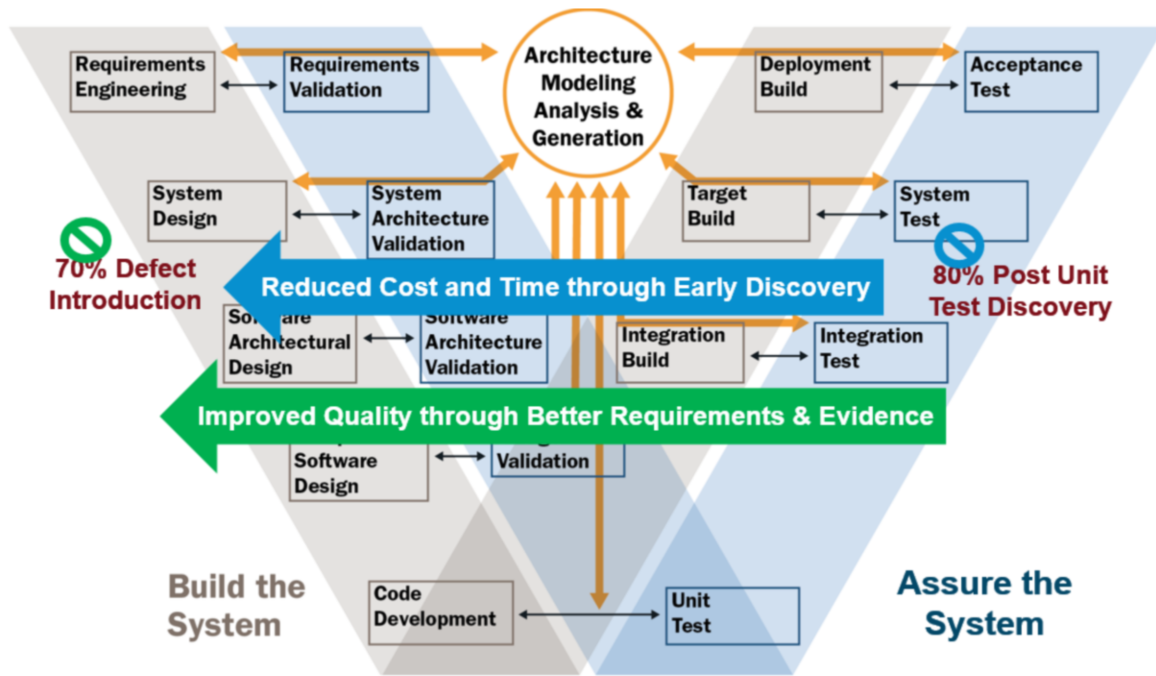


Fig. 8.3: Traditional V-cycle development process model augmented with architecture-centric virtual integration validation activities (reproduced from [205])

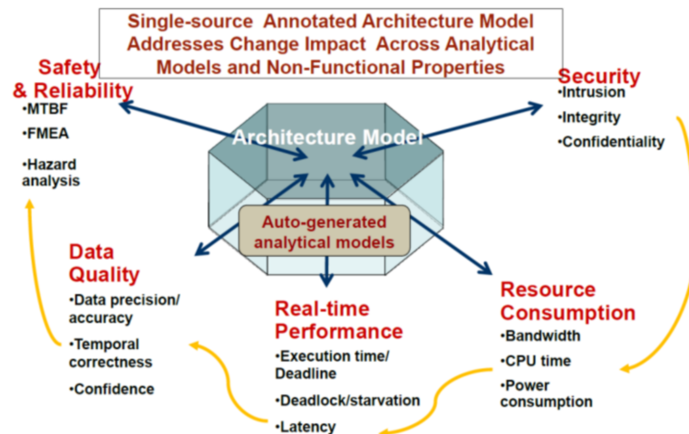


Fig. 8.4: Architecture-centric single source of truth modelling approach (reproduced from [98])

8.2.5 Organisation of the Chapter

The previous sections of this chapter illustrated the central role of ADLs for architecture-centric system architecture virtual integration in order to address the challenges of developing complex CPSs. Therefore, the rest of this chapter introduces ADLs in greater details, and in particular AADL, which is an ADL well suited for the modelling of both the cyber and physical parts of CPSs, and the deployment of the cyber part over the physical execution platform part.

AADL will be introduced in a tutorial-like fashion by presenting the modelling of a simple robot CPS whose purpose is to carry objects in a warehouse. This example is introduced in section 8.4. Even if the AADL language itself does not impose any development process in particular, in order to better illustrate the use of AADL, the aforementioned top down design V-cycle process model augmented with SAVI will be followed (figure 8.3), starting from requirements modelling down to automated code generation from the software architecture model.

For each step of this process, the required AADL constructs will be first introduced and their usage will then be illustrated by modelling the corresponding part of the robot example system.

8.3 AADL Overview

The development of AADL dates back to 1999 when Bruce Lewis working for the US army started a committee to make AADL an SAE standard. Initially standing for Avionics Architecture Description Language, AADL was first developed as an ADL for the avionics domain. It was strongly inspired from another language called MetaH and developed by Honeywell during a DARPA research project. Much of the syntax and the strongly typed property of MetaH (and therefore of AADL) were borrowed from the Ada programming language, which is dedicated to safety-critical embedded systems. As it was soon realized that AADL could be used for any embedded system, it was renamed to Architecture Analysis and Design Language thus preserving the acronym.

As an SAE standard AS-5506, AADL is being developed by the AS2C subcommittee, which includes participants from both academia and industry such as the Software Engineering Institute (SEI) of the Carnegie Mellon university, ISAE, Kansas State University, the U.S Army, NASA, the European Space Agency, INRIA, the Russian Academy of Science, Adventium Labs, Ellidiss Technologies, the Aerospace Corporation, Honeywell, Rockwell-Collins, Airbus industries, Boeing, Dassault Aviation, Toyota and Telecom ParisTech, which is the institution of the authors of this chapter. The committee has also active collaborations with other research initiatives and standardization bodies such as the aforementioned SAVI initiative, the ARINC653 working group and The Open Group Real-time. Despite that the development of AADL was started long time ago, its development is still an ongoing work, for which there has already been 2 major releases while a third one (AADL 3) is in preparation at the time of writing.

A comparison of the capabilities of AADL with other well-known ADLs such as SysML, MARTE and UML is depicted in figure 8.5, according to the intended use and domain covered by the languages. Both SysML, which covers the domain of the physical world and UML, which covers the domain of software were developed mostly for modelling with limited analysis capabilities, due to the weak semantics inherited from their high level of genericity. Conversely, AADL and MARTE were given a stronger semantics due to their more specific covered domain of embedded systems, with AADL performing better on the analysis domain due to its longer history and better maturity[1].

A few approaches have made use of UML and SysML for embedded systems modelling such as respectively Papyrus UML-RT [227] and TTool [274]. However, the only way to make these languages useful for embedded systems is to specialize them using the UML profile extension mechanism. We note however that while the advantage of using UML profiles is that existing tools can be reused with the extended language, it has the drawback of less flexibility in tailoring the language for the domain, including the introduction of accidental complexity due to unnecessary features inherited from the generic language also visible from the extended language. Such is also the case of MARTE, which while showing characteristics similar to those of AADL in terms of intended use and domain suffers from its implementation as a UML profile. The fact that AADL was directly implemented as a DSML avoids this accidental complexity and provides constructs better fitted for the intended domain and use.

While the intended domain of AADL was originally avionics (it was initially named Avionics Architecture Description Language) and soon extended to embedded systems, its version 2 is an adequate language to also model CPSs. Version 2 added abstract components that can be used to model at the system level of abstraction, with similar level of abstraction than SysML. It also introduced annexes for behavior and fault tolerance modelling. In addition, the hardware modelling capability initially developed for embedded systems in combination with the system and abstract components can be reused for modelling the physical parts of CPSs (plant model), with a more appropriate coverage of the domain than what could be done with the too generic SysML block concept. Finally, the software modelling and deployment capabilities of AADL make it an ideal language for addressing the challenges due to the rapidly growing cyber part of CPSs [101].

There are 3 main tools for editing and analysing AADL models. The Eclipse-based Open Source AADL Tool Environment (OSATE) [224], which is developed by the SEI at Carnegie Mellon University, is a reference implementation as it is developed by the leading language architecture team. There is also another Eclipse-based

[1] Note that this comparison taken from [3] is only illustrative and not very precise as one could argue that the UML should somehow overlap with MARTE and AADL regarding the covered domains since all three languages cover parts of the software domain.

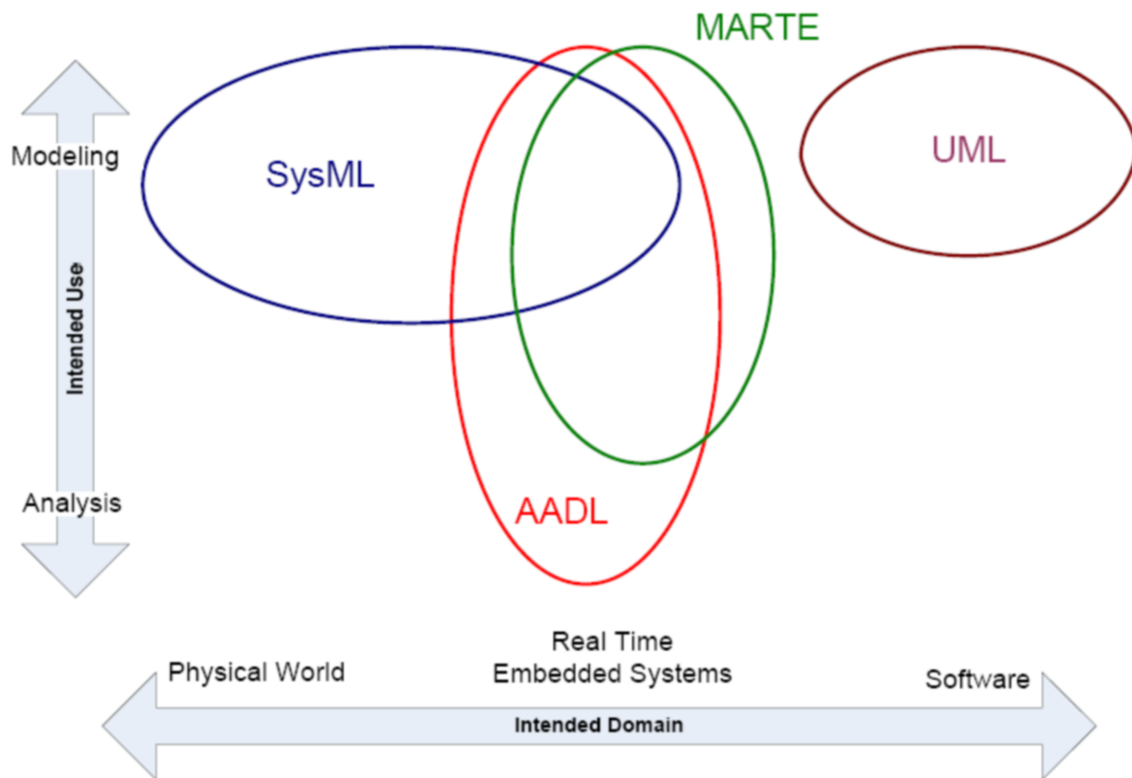


Fig. 8.5: Comparison of well-known ADLs in terms of intended use and domains (reproduced from [3])

tool named MASIW [200], which is developed by the Russian academy of science and finally, commercial tools such as STOOD and AADL Inspector developed by Ellidiss Technologies are also available [7].

8.4 CPS Running Example

The running example that will be used throughout the rest of this chapter consists of a robot system whose purpose is to carry objects in a warehouse from one point to another. The robot shall follow a path indicated by a line drawn on the floor of the warehouse. Several robots may be used in the warehouse and their paths may cross each other. Therefore, the robot should be able to stop momentarily when an obstacle is detected on the line and resume its trajectory once the obstacle is gone. The robot system should be able to carry objects in a minimal amount of time, and the cost of manufacturing such robot should be as low as possible so that the product has advantages over its competitors on the market. For illustration purposes, we will also add a few performance requirements to illustrate the need to perform performances analyses such as timing and latency.

Due to its low cost thus making it easily available, the Lego Mindstorm robot [182] in its NXT version will be used in a configuration as shown on figure 8.6, where two wheel assemblies are provided ❶ with a light sensor for line following ❷ and a sonar for obstacle detection ❸.

The robot will be running a simple line follower application described in [220] for following the edge of a thick black line as illustrated in Figure 8.7. Since the light sensor has a given field of view, the observed light intensity is inversely proportional to the part of the field of view of the sensor occupied by the black line. This light intensity will be used to compute the turn angle of the robot.

The turn angle will be computed using a PID (Proportional Integral Derivative) control algorithm. Such algorithm has the advantage of providing a much smoother line following behaviour compared to the two simpler algorithms depicted in Figure 8.8, since the computed turn angle variable has a much finer grained set of states (theoretically continuous) compared to the left (bang bang) and middle approaches where the turn angle variable has respectively 2 (left or right) and 3 (left, straight or right) states.

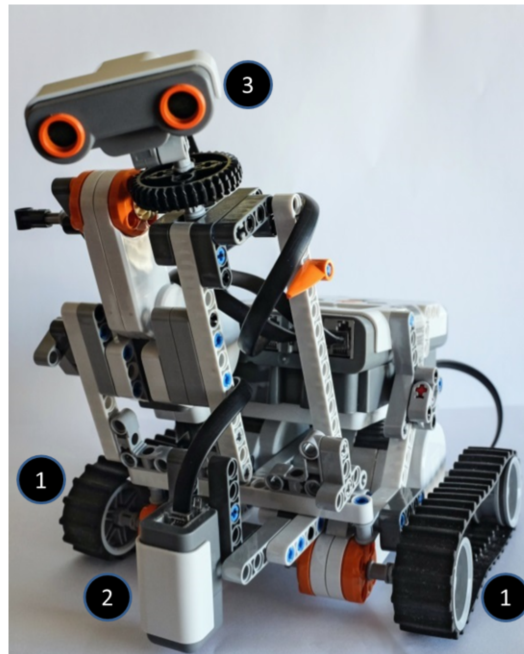


Fig. 8.6: The Lego NXT Mindstorm robot configured to execute a simple line following application with obstacle avoidance

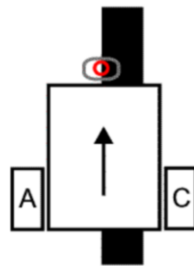


Fig. 8.7: Robot and light sensor following the edge of a line (reproduced from [208])

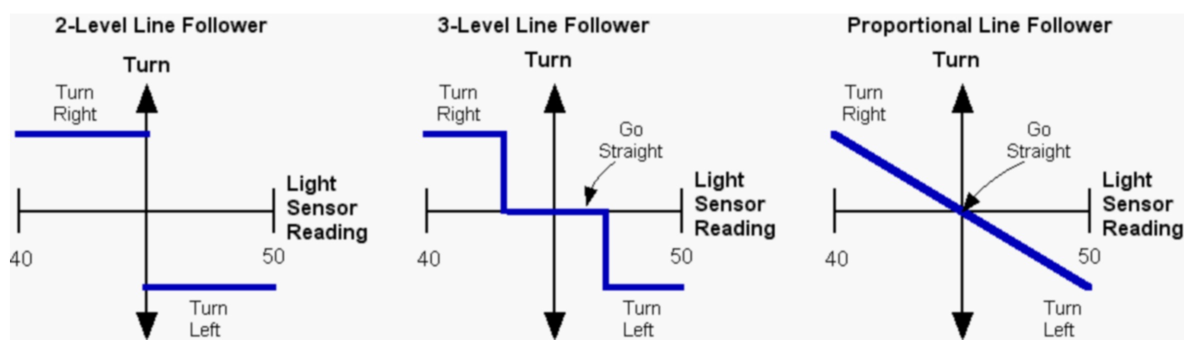


Fig. 8.8: Three ways of controlling the robot with the adopted PID method on the RHS of the figure (reproduced from [208])

8.5 The Development Process

While AADL does not impose any specific development process, it is typical for safety-critical systems to follow a V-cycle development process model augmented with architecture-centric system architecture virtual integration as discussed earlier and illustrated in Figure 8.8. The modelling of the line follower robot CPS will follow such process. Introducing AADL by following a development process allows to better illustrate how AADL can be used and what can be achieved with it.

The architecture-centric virtual integration process is partially supported by the ALISA (Architecture-Led Incremental System Assurance) set of notations and workbench. ALISA is an add-on to the AADL language supported by the OSATE tool that will be used for this tutorial. It originated from the Requirements Definition and Analysis Language [37, 236], which was originally planned to become a requirements annex for AADL.

ALISA augments AADL with a set of languages for the modelling of stakeholders, stakeholder goals, system and software requirements, verification plans and verification methods, and assurance cases for the incremental development of high-assurance systems. Similar to SysML, requirements can be modelled and allocated to architecture elements responsible for verifying them. Requirements can be decomposed and refined incrementally as the architectural design is refined and decisions are made. Complete modelling of assurance cases with arguments and claims linked to requirements is supported making ALISA ideal for the certification of safety-critical systems. Although a thorough description of ALISA is beyond the scope of this chapter, its basic requirements modelling capability will be demonstrated with the running example of this chapter. For instance, a few performance requirements will be captured and linked to the scheduling and latency analyses performed on the running example.

RDAL, and consequently ALISA, were strongly inspired by another work that is worth mentioning in this chapter. It consists of a set of 11 best practices described in the Requirement Engineering Management Handbook (REMH) [184]. The REMH was commanded to Rockwell Collins by the Federal Aviation Administration (FAA) to survey requirements engineering practices in industry [183]. Following this survey, the authors recommended 11 practices for the requirements management of safety-critical systems based on lessons learned from the Requirements Engineering (RE) research and results from the industry survey. The proposed practices can be adopted piecemeal, with minimal disruption of an organizations development process to address the slow or non-existent industry adoption of RE.

For this chapter, the left part of the V-cycle process model decomposed into steps inspired by the best practices of the REMH will be used to illustrate the use of AADL. Such decomposition is depicted in figure 8.9, where the boxes of the central column depicts the steps of the process, with their input and output models shown as boxes respectively located on the left and right sides of the figure. The language used for the models is specified following the ":" symbol. The dashed borders indicate steps that are not covered in this chapter.

The process starts by capturing an overview of the system to be developed creating models of the stakeholders, goals and operational contexts for the system. Such models are then used to derive use cases scenarios for the system, which together with the contexts and goals allow creating a functional architecture for the system. Then, a physical plant model for executing the system functions is specified. Next, the software model is derived from the functional model and deployed on the physical plant. From the deployed system model, analyses such as scheduling and latency can be performed. The models may then be modified to meet performance requirements and optimize non functional properties. Then, an operating system platform is selected to execute the software. The models are then refined for the given operating system using the RAMSES tool [235]. This refined AADL model can then be analysed again providing more accurate results due to specifics of the execution platform taken into account. Depending on the analyses results, the input models at any step of the process may be modified in an iterative manner until the design meets the requirements. Then, code can be automatically generated using RAMSES.

8.6 Modelling the Line Follower Robot with AADL

This section presents the modelling of the running example introduced in section 8.4 by following the development process introduced in section 8.5. Note that for didactic purposes, we will not present the complete set of models but only the parts relevant for the language concepts being taught. For more information, the reader can view the complete set of models made available from [63].

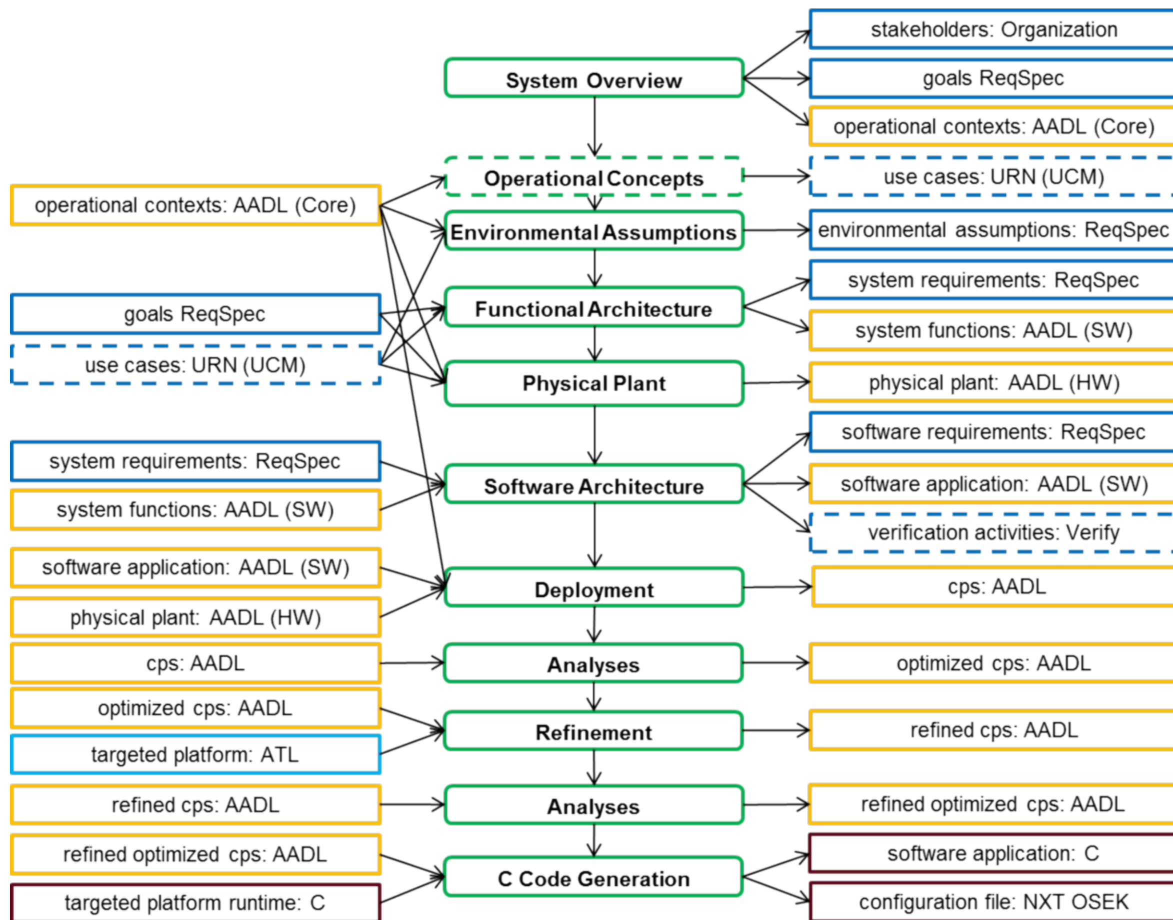


Fig. 8.9: The process for designing the CPS system (left side of the V-cycle of Figure 8.3) and the employed notations

8.6.1 System Overview

The first step of the process consists of specifying the overview of the system to be developed. It includes specifying why the system is needed by stating its purpose and goals, as well as how he system will interact with its environment. Specifying the system and its environment provides a sound understanding of the *boundary* of the system. Such boundary is defined by the set of monitored and controlled variables representing the precise interaction of the system with its environment. Identifying a correct system boundary is difficult since one needs to ensure that all interactions of the system with its environment on which proper system operation depends have been considered, for all the operation modes of the system.

Stakeholders and their Organizations For the running example, let us assume that company *Tartempion Warehouse Equipment* is developing our robot system introduced in section 8.4. Many stakeholders, each of which having their own concerns about the system are involved in the development of the system. Example stakeholders are the customers who will use the system, the designers of the system, the marketing in charge of selling the system, etc.

Stakeholders and their development organisation can be captured with the *organisation* notation of ALISA. An *organisation* has a name and can declare stakeholders. A *stakeholder* also has a name (that must be unique within the containing organisation) and optional description, role, email, phone and supervisor stakeholder. An organisation is declared in a separate file with the extension *org*.

Running Example Specification

The specification of Tartempion Warehouse Equipment Ltd, which develops the line follower objects carrier robot is shown in Listing 14.

```
organization Tartempion_Warehouse_Equipments_Ltd

stakeholder Customer [
    description "The customer of the line follower robot system"
]

stakeholder Marketing [
    description "The people in charge of marketing the line follower
        robot system"
]
```

Listing 14: Stakeholders and their organizations

Stakeholder Goals

As stated in section 8.4, the purpose of the robot is to carry objects in the warehouse by following a predefined trajectory. In addition, it must also be able to stop momentarily upon obstacles detection since other robots working in the same warehouse may be crossing the followed line. Furthermore, the time taken to carry objects should be as low as possible as well as the cost of producing the robot.

Constructs

Goals and requirements modelling is supported by the *ReqSpec* notation of ALISA. A stakeholder goal must have a name and can have several optional attributes such as a title, a description, a category, a rationale, etc. Providing rationale is of particular importance especially for non-trivial goals as it allows to quickly understand why the goal exists. Furthermore, providing rationale avoids questioning the goal over and over again when new people are introduced to the project.

Another important characteristic for goals is traceability to the system that should achieve them. This can be represented in ReqSpec through the *for* construct, which must refer to an AADL classifier representing the system to be built. AADL classifiers will be introduced in section 8.6.1.

Running Example Specification

A specification for the goals of our running example is shown in Listing 15. Goal *G_Behav_1* is first defined, which consists of carrying objects from one point to another by following a trajectory marked on the floor. *G_Behav_1* has *Customer* as stakeholder and has a *Behaviour* category since it relates to the functions of the system we want to build. Note the rationale on *G_Behav_2* that allows understanding why it exists. As will be seen in section 8.6.4, these goals will be transformed into requirements verifiable by the system.

```
stakeholder goals Line_Follower_Robot_Behavior for
   Line_Follower_Functions::Cary_Object [

    goal G_Behav_1 : "Objects_Transportation" [
        description
                "The robot should be able to carry an object between two
                    specified points by following a predefined trajectory
                    in the warehouse."
        stakeholder Tartempion_Warehouse_Equipments_Ltd.Customer
        rationale "This fulfills the main need of customers."
        category Quality.Behavior
    ]
```

```
    goal G_Behav_2 : "Obstacle_Avoidance" [
        description "The robot should be able to avoid obstacles along
           the path."
        stakeholder Tartempion_Warehouse_Equipments_Ltd.Customer
        rationale
                "There may be several robots working on the warehouses
                   and therefore, it is important to avoid damaging the
                   robots and the carried object"
        category Quality.Behavior
    ]
]
```

Listing 15: Behavioural stakeholders goals

Listing 16 shows some performance goals for the system. As opposed to behaviour goals, performance goals can be set with a level of achievement that can be useful when performing design optimisation. Such goals can also be transformed into verifiable requirements that will set bounds on the level of achievement of the goals.

```
stakeholder goals Line_Follower_Robot_Perf for Line_Follower_Robot_Cps::
   Line_Follower_Robot_Cps [

    goal G_Perf_1 : "Minimal Cost" [
        description "The cost of producing the robot should be minimal."
        stakeholder Tartempion_Warehouse_Equipments_Ltd.Customer
           Tartempion_Warehouse_Equipments_Ltd.Marketing
        rationale "The robot should be cheap so that it is easy to market
           ."
        category Quality.Cost
    ]

    goal G_Perf_2 : "Minimal_Transportation_Time" [
        description "The time taken to carry objects should be minimal."
        stakeholder Tartempion_Warehouse_Equipments_Ltd.Customer
           Tartempion_Warehouse_Equipments_Ltd.Customer
        rationale "The robot ."
        category Quality.Performance
    ]
]
```

Listing 16: Quality stakeholders goals

The *for* elements of these goals refer the AADL classifier of the system that is being developed to meet the goal. This is introduced in the following section that presents the modelling of the system, its environment and its contexts of use with AADL.

System, Environment and Contexts of Use

Modelling not only the system but also its environment and their interactions is of primary importance. From these interactions, a set of monitored and controlled variables can be identified. The purpose of the behavioural requirements then consists of specifying the precise relationship between the monitored and controlled variables for all contexts of use of the system and all possible values of the monitored variables.

It is rarely the case that there is only one context of use of a system. For instance, for our running example, the robot carrying an object following a line can be thought of as the normal context of use of the system. But other contexts may exist such as when the robot is under maintenance. Ideally, all contexts of operation of the system should be identified and modelled but for our robot example, we will only present the normal context.

Constructs

The system and its environment can be modelled with AADL using component *types* and their *features*, component *implementations* and their *subcomponents*, component *packages* and component *properties*. Together these constructs form the core of the AADL language.

Component Types and Features (Component Interfaces)

In AADL, a component type declaration is used to provide an interface specifying how a component can interact with other components, without the need to provide details of the component's internal composition. This is achieved by declaring *features*, which are connecting points for connecting the component to other ones. A feature can be typed by a component type declaration. Such typing is used to restrict the connection to other features. A feature must have a direction that can be in, out or in and out. Feature directions also restrict how components can be connected to each other.

Features can be of several kinds, but for modelling at the system level, we only introduce abstract and feature group kinds of features for now. Other kinds of features specific to hardware and software components will be introduced later as they are needed.

An *abstract feature* is a placeholder to be refined to a concrete feature and is to be used for incomplete component type declarations.

A *feature group* is a special kind of feature used to group component features or other feature groups. It is therefore a modelling facility to ease the connection of components with many features, since the connection of a feature group represents the connections of all contained features. Feature group features are declared in a feature group type used for compatibility verification.

Component Implementations (Internal Component Composition)

The internal composition of a component in terms of its subcomponents and their inter connections is specified in AADL with *component implementation* declarations. A component implementation must have an associated component type specifying the component's interaction features as explained above. The advantage of separating component type and implementation declarations is that several implementations can reuse the same type, similar to interfaces of the Java programming language.

Component types and implementations are grouped under the name of component *classifiers*. Such classifiers are used for typing subcomponents in a component implementation. Connections can be declared inside component implementations to connect subcomponents between each other (or to the containing component), according to the features declared in the subcomponent's type.

Component Categories

Components classifiers are divided into 13 categories providing semantics for the domain of embedded systems. These categories are divided into 3 root categories; *composite*, *software* and *execution platform* (hardware).

The composite category is subdivided into two sub-categories:

- *Abstract*: Abstract components are used to represent incomplete components for modelling at a high level of abstraction, similar to SysML blocks, when no specific information is yet known about the kind of component that is modelled. An abstract component implementation can contain subcomponents of any category and can be contained by components of any category. Abstract components can be refined into any of the other AADL concrete component categories, which is useful as more information is known about the system under design.
- *System*: A system represents an assembly of interacting components. A system can have several modes, each representing a possibly different configuration of components and their connectivity contained in the system.

Each component category has its own rule specifying which categories of subcomponents can be contained in a component implementation. This will be detailed along with the *software* and *execution platform* component categories later as we refine the design of the line follower robot example system.

Component Extension and Refinement

Similar to object oriented programming languages, AADL component types and implementations (classifiers) can be extended thus inheriting the features / subcomponents and connections of the extended component. New features / subcomponents and connections can be added to the extending classifier to provide more details. In addition, the type of the inherited features and subcomponents can be *refined*, according to a compatibility rule chosen by the modeller among a set of predefined compatibility rules.

Packages (Organisation of Component Declarations)

Component declarations are contained in *packages*, which can have public and private sections. Package can use declarations from other packages provided that they are imported into the using package.

Properties

Finally, AADL provides a rich sublanguage for modelling *properties*. A specificity of AADL is that property types are defined at the model level and not in the AADL metamodel. This allows users to define their own properties. The AADL standard however defines a set of predefined properties for most common analyses such as timing, scheduling and resource consumption. This avoids everyone defining its own set of properties and contributes to interoperability of the specifications.

A property must have a type (integer, real, range of integer or real values, enumeration, reference, etc.) and optionally a unit type, which can also be defined by users. A property also has an applicability clause restricting the component category or classifier to which property values can be set.

Properties can be set at various places in an AADL specification (e.g.: on a component classifier, a feature, a subcomponent, a connection, etc.). Because component classifiers can be extended, a complex search algorithm must be decided to determine property values. Such algorithm is illustrated in figure 8.10. For example, to determine the value of a property of a subcomponent, the algorithm first search if a value is set on the subcomponent itself (#1). If not, it then searches for a value on its implementation (#2). If no value is found, the extended implementation if it exists is searched (#3). If no value is found, then the component type of the subcomponent is searched for (#4), followed by its extended component if any (#5). If still no value has been found, a value may be searched on the containing component itself (#6), provided that the property type is declared as “inherit”, meaning its value can be inherited by the contained subcomponents. Finally, a property type may be set with a default value, which will then be used if no value has been found by the previous searches.

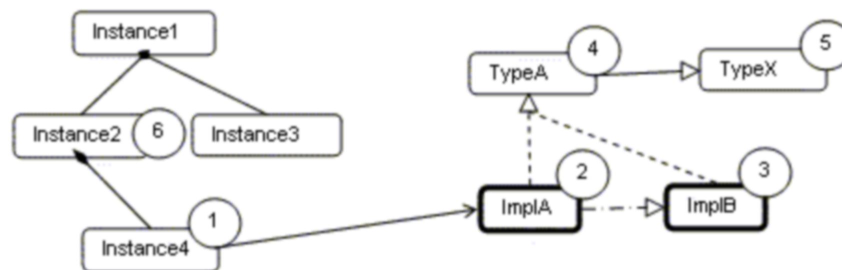


Fig. 8.10: Property value determination (reproduced from [247])

Properties and units are declared in *property sets*, which similar to component packages can be imported to be used by other declarations.

Running Example Specification

The previously introduced core AADL concepts will be understood better by using them to model our system overview, which consists of the system, its environment and its contexts of operation. This will complete the modelling of the system overview step as depicted in the process model of figure 8.9.

We propose a modelling approach for which separate AADL packages stored in different files will be created for the warehouse, the robot physical system, the system functions, the software application and finally the integrated line follower robot CPS. In doing so, AADL component libraries are created independently and become reusable over several projects.

Warehouse

The AADL package declaration for the warehouse is given in Listing 17. The package declares a public section into which two abstract component types are created for the floor and the warehouse. These components of the environment are modelled with the abstract component category, since at this modelling step, it is not meant to know more about these components other than their interaction points. One could also think of using the system category instead of abstract. However, the abstract category is preferred since the AADL system category has the more precise semantics of embedded systems.

An abstract component implementation is declared for the warehouse (line 16), whose name is prefixed (before the dot) by the associated component type (*Warehouse*). It declares a subcomponent for the floor typed by the previously defined *Floor* component type. Note that no component implementation is provided for the floor since its internal composition is of no interest for this modelling. Only the interaction points, which are provided by component types as features are needed in order to model how the system interacts with its environment.

```
package Warehouse
public
    with Physics;

    abstract Floor extends Physics::Reflecting_Object
        features
            applied_force: in feature Physics::Force;
            reacted_force: out feature Physics::Force;
    end Floor;

    abstract Warehouse
        features
            light_source: feature Physics::Light;
    end Warehouse;

    abstract implementation Warehouse.basic
        subcomponents
            floor: abstract Floor;
    end Warehouse.basic;

end Warehouse;
```

Listing 17: Warehouse AADL package

The component type for the warehouse declares a *light_source* abstract feature (line 13 of Listing 17) representing the light emitted by the warehouse lightening system. The feature is typed by a *Light* abstract component type representing the light physical electromagnetic radiation. This component type is declared in a reusable *Physics* package shown in Listing 18. The Physics package is made available to the Warehouse package by adding a *with* clause for the package (line 3 of Listing 17) and by prefixing the imported types by the name of the package that declares them followed by the *::* delimiter symbol (line 7 of Listing 17).

```
package Physics
public

    abstract Reflecting_Object
        features
            light_in: in feature Light;
            light_reflected: out feature Light;
```

```
            sound_in: in feature Sound;
            sound_reflected: out feature Sound;
    end Reflecting_Object;

    abstract Light
    end Light;

    abstract Sound
    end Sound;

    abstract Force
    end Force;

    abstract Power
    end Power;

    abstract Power_Consuming_Object
        features
            power_in: requires bus access Power_Bus;
        properties
            Classifier_Substitution_Rule => Type_Extension;
    end Power_Consuming_Object;

    bus Power_Bus
    end Power_Bus;

end Physics;
```

Listing 18: Physics AADL package

Both the line and the floor component types extend a *Reflecting_Object* component type provided by the Physics package of Listing 18. Since the line follower robot is going to observe the reflected light to follow the trace, the component type provides abstract features for modelling this interaction of the robot with both the floor and the line components. Since a sonar device will also be used by the robot to detect obstacles, features for the reflected sound are also added to the *Reflecting_Object* abstract component type. A *Sound* abstract component type is declared to model the sound physical vibration and used to type these features. Similarly, the floor component type declares features of the *Force* physics type in order to model its interaction with the robot for propulsion.

Robot CPS

Now that classifiers have been declared for the environment of the system (warehouse and the floor), component types are required for the line to follow and for the robot CPS itself. A new package is created for this as shown in Listing 19. Note that here the *system* category is intentionally used for the CPS (line 8) and not the *abstract* one, which was used for components of the environment previously modelled. This is because we want to use the semantics of the AADL system category, since our robot is made of software and hardware components.

```
package Line_Follower_Robot_Cps
public
    with Line_Follower_Software, Robots_Library, Warehouse, Physics,
       Physics_Properties;

    abstract Line extends Physics::Reflecting_Object
    end Line;

    system Line_Follower_Robot_Cps
        features
```

```
            light_sensor_in: in feature Physics::Light;
            sonar_in: in feature Physics::Sound;
            sonar_out: out feature Physics::Sound;
            force_left_wheel: out feature Physics::Force;
            force_right_wheel: out feature Physics::Force;
            force_gripper: out feature Physics::Force;
    end Line_Follower_Robot_Cps;
```

Listing 19: Libe follower CPS AADL package

The *Line_Follower_Robot_Cps* system type declares features for its interaction with its environment such as the *light_sensor_in*, the *sonar_in* and *sonar_out*, the *force_left_wheel* and the *force_right_wheel* and the *force_gripper*, which represents the force on the object held by the gripper of the robot. Note that a single bi-directional feature could have been used to represent the pair of *sonar_in* and *sonar_out* features. However using separate features allows better distinguishing the monitored and controlled variables of the system.

System Context of Operation

After having provided component classifiers for the system and for its environment, those can be instantiated as subcomponents of an enclosing component implementation for the Warehouse. This is for modelling the system and its environment. This is shown in Listing 20 where a component type is created for the warehouse of the robot extending the generic warehouse (line 1).

```
abstract Warehouse_Robots extends Warehouse::Warehouse
    properties
        Physics_Properties::Illuminance => 150.0 lx applies to
           light_source;
end Warehouse_Robots;

abstract implementation Warehouse_Robots.normal extends Warehouse::
   Warehouse.basic
    subcomponents
        line: abstract Line;
        line_follower_robot: system Line_Follower_Robot_Cps;
        obstacle: abstract Physics::Reflecting_Object;
    connections
        floor_robot_light_sensor_in: feature floor.light_reflected ->
           line_follower_robot.light_sensor_in;
        line_robot_light_sensor_in: feature line.light_reflected ->
           line_follower_robot.light_sensor_in;
        obstacle_robot_sonar_in: feature obstacle.sound_reflected ->
           line_follower_robot.sonar_in;
        robot_sonar_out_obstacle: feature line_follower_robot.sonar_out
           -> obstacle.sound_in;
        force_left_wheel_floor: feature line_follower_robot.
           force_left_wheel -> floor.applied_force;
        force_right_wheel_floor: feature line_follower_robot.
           force_right_wheel -> floor.applied_force;
        light_source_line: feature light_source -> line.light_in;
        light_source_floor: feature light_source -> floor.light_in;
        Warehouse_Robots_normal_new_connection: feature light_source ->
           obstacle.light_in;
    properties
        Physics_Properties::Curvature_Radius => 99.0 mm applies to line;
end Warehouse_Robots.normal;
```

Listing 20: Normal context of operation of the robot CPS

A property value is set to characterise the *illuminance* of the light source in *lx* units (line 3). The property is declared to apply to the *light_source* feature inherited from the extended *Warehouse* classifier. An abstract component implementation (line 6) is created for modelling the normal context of use of the robot CPS, that is when it is carrying an object. It extends the generic warehouse component containing the floor subcomponent by adding other subcomponents for the line to follow, the robot CPS and an obstacle object.

One problem that is faced with the textual representation of Listing 20 is the difficulty of perceiving how subcomponents interact with each other. One advantage of AADL is that it proposes two notations; a textual and a graphical one. Therefore, users can choose the notation that is most appropriate depending on what is being viewed. Only the textual notation has been used so far, which was appropriate to display component types. However for component implementations containing connected subcomponents, the graphical notation is more appropriate. Therefore, a diagram of the AADL graphical notation is shown in figure 8.11 for the normal operation of the robot CPS. This diagram is obviously much easier to read than the code of Listing 20. However, the diagram does not show all information. For instance, the illuminance property of the warehouse is not visible on the diagram. The diagram is just a view while the textual always contain all information of the model.

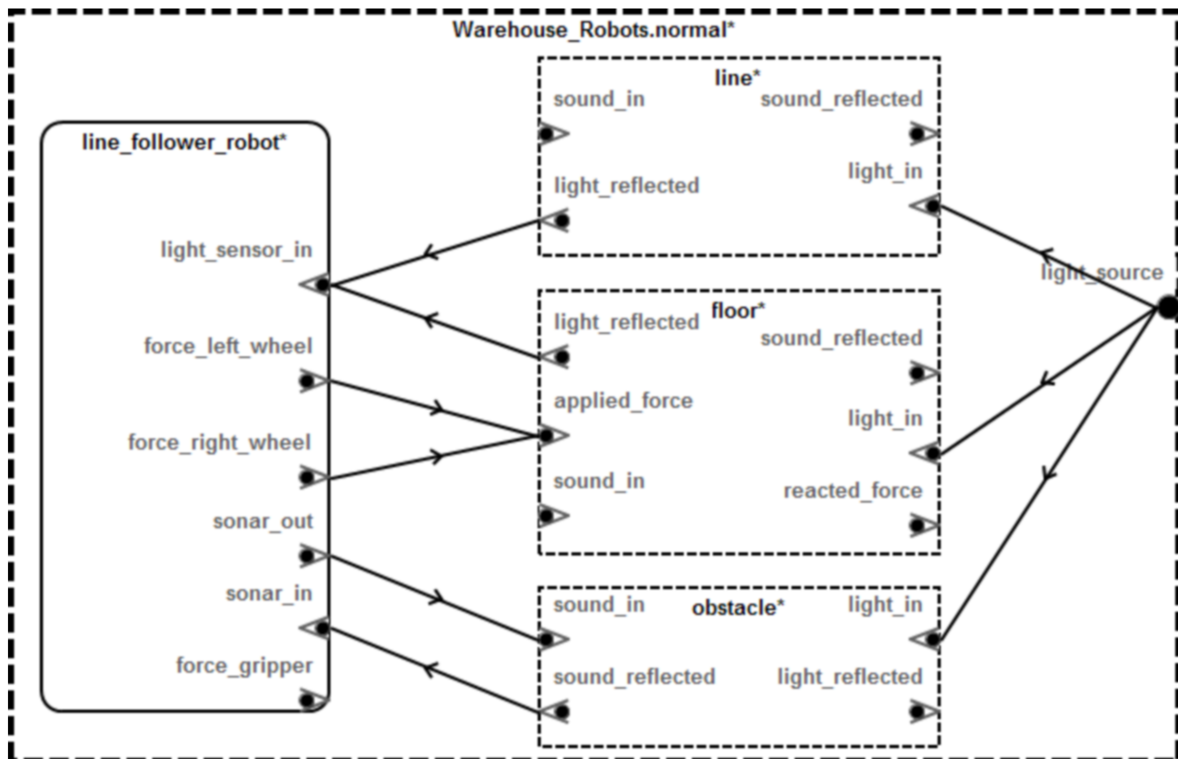


Fig. 8.11: The normal context of operation of the robot CPS displayed using the AADL graphical notation

In order to better interpret the diagram, figure 8.12 summarises the AADL graphical notation for component categories. Systems are represented as rounded box while abstract components as dashed rectangles. Abstract features are represented by the symbol ">" with a dot inside and connections as lines with arrows indicating the direction of the represented flow.

The diagram of figure 8.11 can be called a context diagram as it describes the context of operation of the system and clearly distinguishes it from its environment. This distinction is achieved by the different graphical representations of system and abstract components, which further justifies the choice of modelling the system to be built with the system category as opposed to abstract for the other entities of the environment.

The diagram also shows how the robot system contained in the warehouse interacts with its environment, represented by abstract features and connections for sensing the obstacle through the sonar, and the floor and line to follow through the light sensor. The interactions of the wheels with the floor for propulsion are also represented. From this context diagram, the system boundary is precisely identified for the given context as being the set of monitored (in) and controlled (out) features of the robot system that are *connected* to entities of

the environment. Also note that for this context of operation, the *force_gripper* feature is not connected as it is assumed that the carried object is contained by the robot system and is therefore not part of the environment. For a different context for which the robot is picking the object, that object would then be part of the environment as it would not yet have been picked by the robot but would be interacting with the robot's gripper actuator. In this context, the *force_gripper* feature would then be connected to the object to be carried.

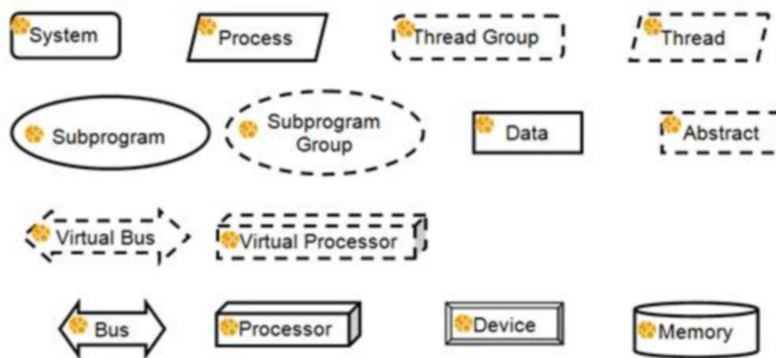


Fig. 8.12: The AADL graphical notation for component categories

When developing a system, it is often challenging to be aware of all the interactions the system may have with its environment. One must not forget interactions that must be taken into account for safe operation of the system in the given context and decide which ones can be neglected. A context diagram can help identifying the neglected interactions by searching for features of the system and components of its environment of the same type that are not connected, thus indicating a neglected interaction. For instance, such is the case of the *light_reflected* and *light_sensor_in* features of the obstacle and the robot CPS, which remain unconnected. Although the obstacle actually does reflect the light from the warehouse, its interaction with the light sensor of the robot is neglected, as the light sensor will be close enough to the line and not be influenced by this light. Such neglecting can also be modelled as a verifiable environmental assumption constraining the environment of the system. Environmental assumptions modelling will be presented in section 8.6.3.

Note that only the line following normal context of operation is presented here. Another context could be modeled for when the robot is under maintenance. This context would be modeled as another abstract component implementation for the same warehouse component type, where only the interacting subcomponents and connections for the context would be instantiated.

8.6.2 Operational Concepts

The next step in the modelling process of figure 8.9 consists of developing scenarios that describe how the system will be used in the contexts provided by the system overview. Use cases are a good way to do this. Several languages exist for use case modelling. The Use Case Maps sublanguage of the User Requirements Notation [155] and its Eclipse-based tool jUCMNav [1] are well suited for this, since the modeled use cases and scenarios can be simulated for their validation. This turned out to be extremely useful in the modelling of the isolette thermostat example provided by the REMH [184] as several errors were discovered in the natural language use cases. However, presenting use case modelling is beyond the scope of this chapter and the reader can refer to [38, 37] for more information.

8.6.3 Environmental Assumptions

The introduction of this chapter on AADL pointed out how well known design faults leading to catastrophic errors such as the Ariane 5 bug were due to mismatched assumptions between a system and its environment.

Therefore, identifying the environmental assumptions on which a system depends for correct operation is essential for reusing the system in different contexts and avoiding misuse of the system.

Assumptions can specify conditions that must be met by one or several entities of the environment of the system for proper operation. They can also constrain the types, ranges, and units of the monitored and controlled variables of the system.

Let us assume that the robot CPS is using a simple passive light sensor, which measures the light reflected by the line and the floor in its field of view. An obvious environmental assumption is that a minimum quantity of light is emitted by the light source of the warehouse. This is essential for proper operation of the line following robot otherwise it cannot see the line to follow.

Another example assumption, would be to impose a limit on the curvature radius of the line to follow, since there is a limit on the steering capability of the robot given its speed and the carried mass. A limit on the mass of the carried object would also be a reasonable assumption.

Constructs

Environmental assumptions are modelled using the *ReqSpec* notation as *system requirements*, but however assigned to elements of the environment and not to the system itself. Similar to goals, system requirements can have a *description*, a *rationale*, a *category*, etc. Furthermore, a requirement can also be traced to a goal that it transforms into a verifiable requirement using the keyword *see*.

Requirements must be verifiable by the entities they constrain. In order to achieve this, they can be set with a predicate expression or a verification activity that can be registered in the workbench and referred by the requirement. A predicate expression can for instance compare the value of properties on the architecture model with bounds set for those values and captured in the requirements set.

Running Example Specification

Listing 21 shows the modelling of environmental assumptions for the robot carrier CPS with the ReqSpec notation. A system requirements specification named *Line_Follower_Robot_Env_Assumptions* is created and assigned via the *for* keyword to the *Warehouse_Robot.normal* system implementation representing the normal context of operation of the robot (see figure 8.11). Both assumptions are captured as system requirements but constraining the *light_source* and the *line* subcomponents of the warehouse as defined by the *for* keyword (lines 3 and 11). Such assumptions are set with the *Kind.Assumption* category to further distinguish them from the system's requirements.

```
system requirements Line_Follower_Robot_Env_Assumptions for
   Line_Follower_Robot_Cps::Warehouse_Robots.normal [

    requirement EA_1: "Minimum Warehouse Luminosity" for light_source [
        description "The power of the light source shall not be less than
             the Minimum Illuminance value"
        rationale "Otherwise the light sensor of the robot will not be
            able to give proper readings given its sensitivity and its
            calibration that was performed under these conditions of
            minimum lightening. Study my_study has shown that the value of
             100 lux is a lower bound for warehouses illuminance"
        category Kind.Assumption
        val Minimum_Illuminance = 100.0 lx
        value predicate #Physics_Properties::Illuminance >=
            Minimum_Illuminance
    ]

    requirement EA_2: "Minimum Curvature Radius" for line [
        description "The curvature radius of the line to be followed by
            the robot shall not be lower than the specified
            Minimum_Curvature_Radius value"
        rationale "Otherwise the robot given its speed, mass and response
             time will not be able to follow the line."
        category Kind.Assumption
        val Minimum_Curvature_Radius = 100.0 mm
```

```
        value predicate #Physics_Properties::Curvature_Radius >=
            Minimum_Curvature_Radius
    ]
]
```

Listing 21: Environmental assumptions for the robot CPS

Value predicates are set to the assumptions for their automated verification. For example, the predicate of *EA_1* (line 8) compares the value of the *Illuminance* property of the *light_source* feature of the warehouse to a minimum value of 100.0 lx declared in the requirement (line 7).

Note that if an entity constrained by the assumption were to be developed by another organisation, then the organisation would provide an extension of the abstract component representing the entity and the assumption would then be converted to an equivalent requirement for the entity.

8.6.4 Functional Architecture

The next step of the followed development process is to develop the functional architecture. This step assumes that the operational concepts have been developed as mentioned in section 8.6.2 through use case modelling. Developing the functional architecture then consists of providing a first set of system functions identified from the use cases scenarios.

Following the architecture-led ALISA approach, high level requirements are first modeled for the system functions. Then, their corresponding architecture model elements are specified. Finally, the requirements are assigned to these architecture elements. This is an iterative process where design decisions are taken on the decomposition of the architecture of the system into subcomponents and the initial high level requirements are decomposed accordingly.

System Requirements

Constructs

Like for assumptions, requirements are modelled with the ReqSpec notation. Requirements modelling has already been introduced during the modelling of environmental assumptions. Additional properties needed for this modelling are presented below:

- *decomposes*: refers to one or more requirements that this requirement decomposes or is derived from. The decomposing requirement is assigned to an architecture component contained in a component to which the decomposed requirement is assigned to.
- *see goal*: refers to one or more stakeholder goals that the requirement represents, typically when the referred goal is transformed into this verifiable requirement.

Running Example Specification

The high level stakeholder goals of Listing 15 transformed into verifiable requirements are shown as requirements *R_Behav_1* and *R_Behav_2* of Listing 22. The *see goal* clause of line 6 refers to the stakeholder goals of Listing 15.

```
system requirements Line_Follower_Robot_Behavior for
    Line_Follower_Functions::Cary_Object.basic [

     requirement R_Behav_1 : "Carry_Object_Function" [
         description
                 "The robot shall carry an object between two specified
                     points by following a predefined trajectory in the
                     warehouse."
```

```
        see goal Line_Follower_Robot_Behavior.G_Behav_1
        category Quality.Behavior
    ]

        requirement R_Behav_1_1 : "Pick_Up_Object_Function" for
           pick_up_object [
            description "At the beginning of the path, the robot shall
               pick up an object on the floor."
            category Quality.Behavior
            decomposes R_Behav_1
        ]

        requirement R_Behav_1_2 : "Follow_Line_Function" for follow_line
           [
            description "The robot shall follow a line on the floor of
               the warehouse."
            category Quality.Behavior
            decomposes R_Behav_1
        ]

        requirement R_Behav_1_3 : "Drop_Off_Object_Function" for
           drop_off_object [
            description "At the end of the path, the robot shall drop off
                the carried object on the floor."
            category Quality.Behavior
            decomposes R_Behav_1
        ]

    requirement R_Behav_2 : "Avoid_Obstacles_Function" for
       detect_obstacle [
        description
                "The robot shall avoid colliding with obstacles along the
                    path."
        see goal Line_Follower_Robot_Behavior.G_Behav_2
        category Quality.Behavior
    ]
]
```

Listing 22: High level system requirements

Those two high level requirements are further decomposed into sub-requirements that each must be verified for the high level requirement to be verified. For instance, *R_Behav_1* is decomposed into *R_Behav_1_1*, *R_Behav_1_2* and *R_Behav_1_3* (Listing 22), which state that carrying an object between two points is achieved by picking up the object, moving it along a path and dropping it at destination. This is specified by the *decomposes* keyword referring to the decomposed requirement.

Architecture

Following the architecture-lead modelling approach, the system functions must be represented in the system architecture. This is performed again using the *abstract* component category, which was also used for modelling the components of the environment. Since the abstract category has already been introduced for modelling the system in its environment, we can proceed directly to the running example specification below.

Running Example Specification

System functions are shown in Listing 23. An abstract component type is first declared for the *Carry_Object* function, and a following implementation defines subcomponents for the *pick_up_object*, *follow_line*, *drop_off_object*

and *detect_obstacles* sub-functions. Abstract component types are declared for each of these sub-functions with their features representing function variables. An abstract component is declared for the type of a robot state variable determining if an obstacle has been detected. The *detect_obstacles* function will output such variable and pass it to the *follow_line* function that will stop the robot in case an obstacle is detected. This is declared by the connection of line 15 in Listing 23.

```
package Line_Follower_Functions
public
    with Physics;

    abstract Cary_Object
    end Cary_Object;

    abstract implementation Cary_Object.basic
        subcomponents
            pick_up_object : abstract Pick_Up_Object.basic;
            follow_line: abstract Follow_Line.basic;
            drop_off_object: abstract Drop_Off_Object.basic;
            detect_obstacle: abstract Detect_Obstacle;
        connections
            obstacle_detection_state: feature detect_obstacle.state ->
               follow_line.state;
    end Cary_Object.basic;
```

Listing 23: High level architecture system functions

Now that a component type has been provided for the global carry object system function, the stakeholder goals of Listing 15 can be assigned to this component with the *for* clause of the goals package. Note that the component type is used for the assignment instead of the component implementation because of the visibility of requirements and goals in ALISA. Similar to property visibility (see section 8.6.1), a goal assigned to a component type is automatically assigned to all implementations and subcomponents of the type and to the extending classifiers and their subcomponents, and so on.

Similarly, the top level requirements created from the goals are assigned to the *Carry_Object.basic* component implementation (Listing 22) and each decomposing requirement is assigned to the corresponding subcomponent decomposing the main component.

Decomposing the Follow Line Function

As an example of further requirements and architecture decomposition, the abstract component of the *follow_line* function of Listing 23 is detailed in Listing 24. The *Follow_Line* component type declares an input feature for the light intensity and output features for the left and right wheel power. Such features are typed by the abstract component types of the Physics package (Listing 18). A data access to a robot state internal variable is further added. This variable will be set by the obstacle detection subprogram to stop the robot when an obstacle is detected.

```
abstract Follow_Line
    features
        light_intensity: in feature Physics::Light;
        left_motor_power: out feature Physics::Power;
        right_motor_power: out feature Physics::Power;
        state: feature Robot_State;
end Follow_Line;
```

Listing 24: Follow Line architecture system functions

The line following sub function is further decomposed into two sub functions as illustrated by the the *Follow_Line.basic* component implementation shown in Listing 25, and by its corresponding diagram of figure 8.13. The implementation specifies subcomponents for two functions for computing the turn angle of the robot as a function of the measured light intensity and to compute the wheels motor power from the desired angle.

```
abstract implementation Follow_Line.basic
    subcomponents
        compute_turn_angle: abstract Compute_Turn_Angle.pid;
        compute_wheels_motors_power: abstract Compute_Wheels_Motors_Power
           .basic;
    connections
        light_intensity_compute_turn_angle: feature light_intensity ->
           compute_turn_angle.light_intensity;
        turn_angle_compute_wheels_power: feature compute_turn_angle.
           turn_angle -> compute_wheels_motors_power.turn_angle;
        compute_wheels_motors_power_left_motor_power: feature
           compute_wheels_motors_power.left_motor_power ->
           left_motor_power;
        compute_wheels_motors_power_right_motor_power: feature
           compute_wheels_motors_power.right_motor_power ->
           right_motor_power;
        state_compute_wheels_motors_power: feature state ->
           compute_wheels_motors_power.state;
end Follow_Line.basic;
```

Listing 25: Line following architecture system function implementation

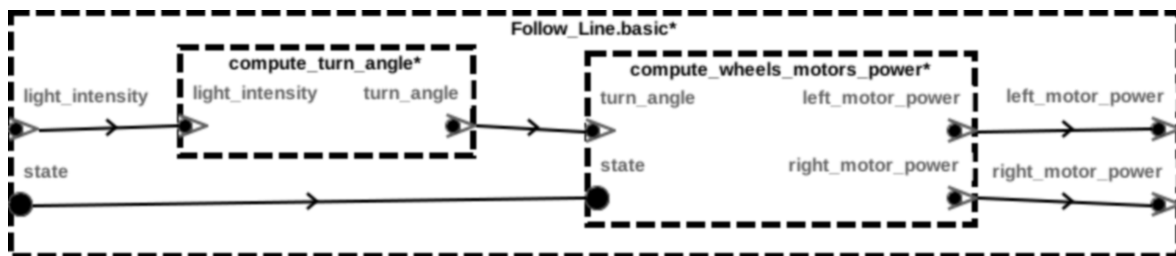


Fig. 8.13: The decomposition of the line following function into sub functions

Feature connections connect the subcomponent functions to the features of the enclosing component and the features between each contained subcomponent. The graphical representation of figure 8.13 clearly illustrates the dependency between the two sub functions, which is a practice recommended by the REMH.

On the requirements side, a new requirements set is created for the *Follow_Line.basic* component implementation as shown in Listing 26. It contains requirements decomposing *R_Behav_1_2* into requirements *R_Behav_1_2_1*, *R_Behav_1_2_2* and *R_Behav_1_2_3*. These requirements require setting the values of the turn angle internal variable and the left and right wheel power controlled variables. These requirements are allocated via their *for* clauses to the *compute_turn_angle* and the *compute_wheels_motors_power* subcomponents having the aforementioned variables as output features.

```
system requirements Follower_Line_Behavior for Line_Follower_Functions::
   Follow_Line.basic [

    requirement R_Behav_1_2_1 : "Set_Turn_Angle_Function" for
       compute_turn_angle [
        description "The controller shall set the value of the turn angle
            variable."
        category Quality.Behavior
        decomposes Line_Follower_Robot_Behavior.R_Behav_1_2
    ]
```

```
    requirement R_Behav_1_2_2 : "Set_Left_Wheel_Power_Function" for
        compute_wheels_motors_power [
         description "The left wheel controller shall set the value of the
              left wheel power variable."
         category Quality.Behavior
         decomposes Line_Follower_Robot_Behavior.R_Behav_1_2
    ]

    requirement R_Behav_1_2_3 : "Set_Right_Wheel_Power_Function" for
        compute_wheels_motors_power [
         description "The right wheel controller shall set value of the
            right wheel power variable."
         category Quality.Behavior
         decomposes Line_Follower_Robot_Behavior.R_Behav_1_2
    ]
]
```

Listing 26: Sub-requirements for the sub-functions of the line following system function

8.6.5 Physical Plant Model

A functional architecture model has been provided with associated requirements created from the original stakeholder goals of the system overview. Such model remains quite abstract and needs to be refined by providing a software model and an execution platform model for execution of the software. These models will allow more accurate analyses and verification since many analyses strongly depend on the plant (hardware) model providing execution resources for the system functions. Therefore, the next step is to provide a detailed model of the robot hardware plant that will be considered to execute the system functions in order to meet the system requirements.

In the following, the modelling of a simple Lego NXT Mindstorm robot is first introduced. A Lego robot is a highly configurable system and the first step is to model a specific configuration suitable for achieving the system goals. The modelling of this configuration will allow briefly introducing how component families can be modelled in AADL, for which more information can be found in [99].

Constructs

The composite component categories, which consists of the *system* and *abstract* sub-categories have been introduced in section 8.6.1 and were used to model the system overview. Abstract components were also used to model a first set of system functions. The execution platform (hardware) component sub-categories are now presented in order to use them to model the plant. They are decomposed into six sub-categories:

- *Processor*: A processor is an abstraction of hardware and software responsible for scheduling and executing threads, as well as virtual processors representing processor partitions such as ARINC653. Processors may contain memories to model a processor's internal structure such as caches. A processor can access memories and devices via bus access features.
- *Virtual Processor*: A virtual processor represents a logical resource that is capable of scheduling and executing threads and other virtual processors bound to it. Such component is useful to model partitioned processors such as ARINC653 for integrated modular avionics. Virtual processors can be declared as subcomponents of a processor or of another virtual processor. They are implicitly bound to the processor or virtual processor that contains them. They can also be used to represent hierarchical schedulers.
- *Memory*: A memory is a component for storing code and data. Memories can represent randomly accessible physical storage such as RAM, ROM, or more permanent storage such as disks or logical storage. Subprograms, data and processes are bound to memory components for being accessed by processors executing threads.
- *Bus*: A bus is a component that can exchange control and data between memories, processors and devices. It is an abstraction of a communication channel and associated communication protocols. Communication protocols can also be explicitly modelled with virtual buses.

- *Virtual Bus*: A virtual bus is a logical bus abstraction such as a virtual channel or communication protocol.
- *Device*: A device is a dedicated hardware performing built-in function(s) and acting as an interface to the physical world of CPSs. Devices can be used for both hardware and software parts of systems. Hardware sensors and actuators are modelled as devices, which can physically be connected to processors via buses. For software, devices can be logically connected to application software components thus representing the software part of corresponding hardware devices, such as drivers residing in a memory and executed on an external processor.

Running Example Specification

The NXT Lego robot includes plastic blocks that can be assembled to create the robot physical structure. It also includes two motors and wheels assemblies, a light sensor, a sonar sensor and a so-called brick. The brick itself contains several components such as a battery, an ARM7 processor containing a RAM memory, a Bluetooth module, input and output circuits, etc. An informal diagram of the NXT brick obtained from the NXT hardware developer kit [2] is shown in figure 8.14.

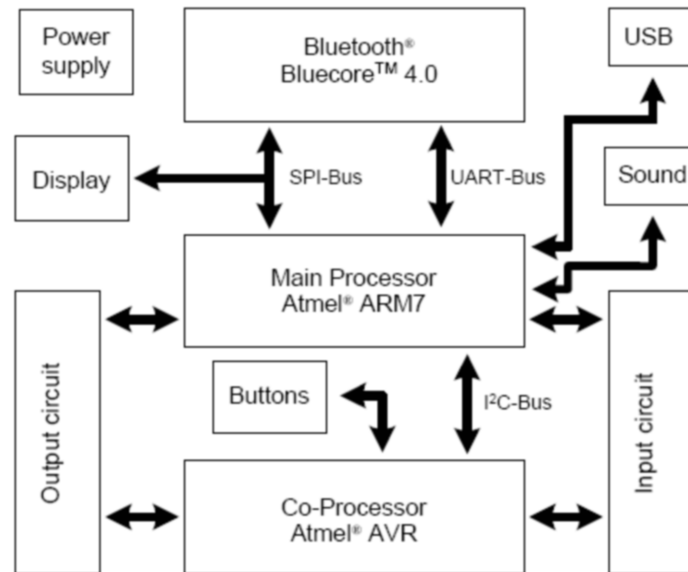


Fig. 8.14: An informal diagram of the NXT brick (reproduced from [2])

In order to model these elements in AADL, a *Robots_Library* package (Listing 27) is first created to contain classifiers for being instantiated to specify the configuration of the Lego NXT robot. A system type (line 5) and a system implementation (line 10) are declared for the robot. At this level of abstraction, only a single subcomponent for the brick is contained in the robot's system implementation (line 12). This is because it can be can assumed that any robot will always contain a brick while the presence of other components will depend on the specific hardware configuration selected for the robot according to its mission.

System classifiers are then declared for the brick subcomponent (lines 15 and 20). For the purpose of this chapter, only a simplified version of the brick will be modeled, which is sufficient for the analyses that will be presented in this chapter. Such version is limited to the modelling of the processor, the power supply, the input and output circuit and buses to connect these subcomponents.

The basic implementation of the generic brick is illustrated in line 20 of Listing 27 and in the diagram of figure 8.15. Power consumption is a concern for robots as it determines their autonomy. Therefore, the electrical power supply is modelled and represented as a device component connected to a power bus also connected to the processor. Note that for this hardware model, bus access and bus access connections are used instead of abstract features such as those of the system overview. Having explicit components for buses allows representing concurrency on communication resources. In addition, logical connections modelled in the software application can be mapped to hardware bus components to estimate resources consumption.

```
package Robots_Library
public
    with OSEK, Physics, Physics_Properties, SEI;

    system Robot
        properties
            Classifier_Substitution_Rule => Type_Extension;
    end Robot;

    system implementation Robot.basic
        subcomponents
            brick: system Brick.basic;
    end Robot.basic;

    system Brick
        properties
            Classifier_Substitution_Rule => Type_Extension;
    end Brick;

    system implementation Brick.basic
        subcomponents
            main_processor: processor Generic_Processor;
            power_supply: device Generic_Battery;
            power_bus: bus Physics::Power_Bus;
        connections
            power_supply_power_bus: bus access power_supply.power ->
               power_bus;
            power_bus_main_processor: bus access power_bus ->
               main_processor.power_in;
    end Brick.basic;

    processor Generic_Processor extends Physics::Power_Consuming_Object
        features
            data_bus: requires bus access Data_Bus;
    end Generic_Processor;

    bus Data_Bus
        properties
            Transmission_Time => [Fixed => 10 ns .. 100 ns; PerByte => 10
                ns .. 40 ns;];
    end Data_Bus;

    device Generic_Battery
        features
            power: requires bus access Physics::Power_Bus;
    end Generic_Battery;
```

Listing 27: AADL package for a robot components library

This component for the brick is generic in the sense that it has nothing specific to the NXT version. In order to capture these specificities, an extension of the brick system implementation is provided as illustrated by Listing 28 and the corresponding diagram of figure 8.16. The system type *Nxt_Brick* extends the previously defined *Brick* system type and adds bus access features (lines 3 to 9) for the brick to be connected to the sensors and actuators of the robot. The corresponding implementation *Nxt_Brick.basic* (line 12) extends the *Brick.basic* implementation and adds subcomponents specific to the NXT brick for the input and output circuits and a data bus to connect them to the processor. The input and output circuits take data from the processor as input and

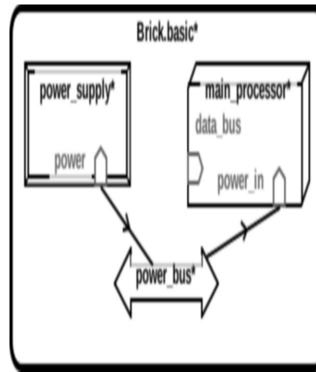


Fig. 8.15: A basic system implementation for a brick of a Lego Mindstorm robot

read / send data accordingly to the connected sensors and actuators. Device component types are provided for the input and output circuits (lines 40 and 49), which inherit their *power_in* bus access feature from the extended *Power_Consuming_Object* provided by the Physics package.

```
system Nxt_Brick extends Brick
    features
        in_1: requires bus access Sensor_Bus;
        in_2: requires bus access Sensor_Bus;
        in_3: requires bus access Sensor_Bus;
        in_4: requires bus access Sensor_Bus;
        out_1: requires bus access Actuator_Bus;
        out_2: requires bus access Actuator_Bus;
        out_3: requires bus access Actuator_Bus;
end Nxt_Brick;

system implementation Nxt_Brick.basic extends Brick.basic
    subcomponents
        main_processor: refined to processor Arm_Processor.nxt;
        input_circuit: device Input_Circuit;
        output_circuit: device Output_Circuit;
        data_bus: bus Data_Bus;
    connections
        brick_input_circuit_1: bus access in_1 -> input_circuit.in_1;
        brick_input_circuit_2: bus access in_2 -> input_circuit.in_2;
        brick_input_circuit_3: bus access in_3 -> input_circuit.in_3;
        brick_input_circuit_4: bus access in_4 -> input_circuit.in_4;
        brick_output_circuit_1: bus access output_circuit.out_1 -> out_1;
        brick_output_circuit_2: bus access output_circuit.out_2 -> out_2;
        brick_output_circuit_3: bus access output_circuit.out_3 -> out_3;
        power_bus_input_circuit: bus access power_bus -> input_circuit.
           power_in;
        power_bus_output_circuit: bus access power_bus -> output_circuit.
           power_in;
        main_processor_data_bus: bus access main_processor.data_bus ->
           data_bus;
        data_bus_input_circuit: bus access data_bus -> input_circuit.
           control;
        data_bus_output_circuit: bus access data_bus -> output_circuit.
           control;
    properties
```

```
        SEI::PowerCapacity => 9.0 W applies to power_supply; -- 6 AA
           batteries in series at 1 A
        Physics_Properties::Mass => 200.0 g;
        Scheduling_Protocol => (RMS) applies to main_processor;
        OSEK::SystemCounter_MaxAllowedValue => 2000 applies to
           main_processor;
        OSEK::SystemCounter_TicksPerBase => 1 applies to main_processor;
        OSEK::SystemCounter_MinCycle => 1 applies to main_processor;
end Nxt_Brick.basic;

device Input_Circuit extends Physics::Power_Consuming_Object
    features
        in_1: requires bus access Sensor_Bus;
        in_2: requires bus access Sensor_Bus;
        in_3: requires bus access Sensor_Bus;
        in_4: requires bus access Sensor_Bus;
        control: requires bus access Data_Bus;
end Input_Circuit;

device Output_Circuit extends Physics::Power_Consuming_Object
    features
        out_1: requires bus access Actuator_Bus;
        out_2: requires bus access Actuator_Bus;
        out_3: requires bus access Actuator_Bus;
        control: requires bus access Data_Bus;
end Output_Circuit;
```

Listing 28: NXT brick extension

The generic *main_processor* subcomponent of the extended generic brick system implementation is *refined to* an *ARM_Processor.nxt* type as shown in Listing 28 (line 14). The type of this refined subcomponent is an extension of the generic processor component type as shown in Listing 29 below (line 1). It contains a property association setting the MIPS capacity of the processor (line 3). The processor implementation provides a RAM memory subcomponent (line 8) with a memory capacity property value of 64 KByte (line 10) as given by the Lego hardware specification of the NXT brick.

```
processor Arm_Processor extends Generic_Processor
    properties
        SEI::MIPSCapacity => 43.2 MIPS;
end Arm_Processor;

processor implementation Arm_Processor.nxt
    subcomponents
        ram: memory RAM_Memory;
    properties
        SEI::RAMCapacity => 64.0 KByte applies to ram;
end Arm_Processor.nxt;

memory RAM_Memory
    features
        data_bus: requires bus access Data_Bus;
end RAM_Memory;
```

Listing 29: ARM processor for the NXT brick extension

Such combination of classifier extensions and subcomponent and feature refinements is extremely useful in modelling component families as further explained in [99]. In this modelling, two different ways are used to provide the NXT-specific information. One of them is to add property associations within the extending

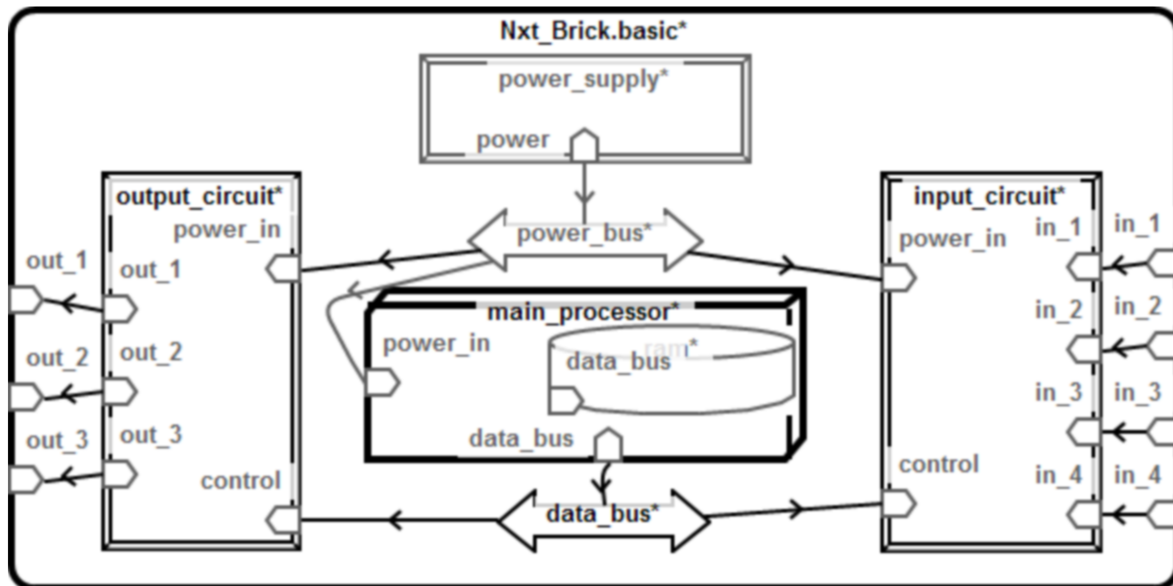


Fig. 8.16: The internal composition of a NXT Lego Mindstrom brick

component implementation of the brick with values pertaining to this specific model, as shown for the power capacity value set for the power supply subcomponent (line 32 of Listing 28).

The second way is to refine the classifier of a subcomponent as is done for the ARM processor extending the generic processor. Having this new classifier then allows providing more details such as the internal composition of the processor, which includes a RAM memory subcomponent. MIPS and RAM capacity property values can be declared in theses processor extensions (lines 3 and 10 of Listing 29). Deciding on which mechanism should be used depends on the modelling context. For example, the advantage of creating such extension classifiers for the processor is that they allow encapsulating the NXT specific information and makes it easy to reuse in other specifications.

Now that component classifiers are provided for the brick, other classifiers must be declared for the sensors and actuators of the robot. Those are presented in Listing 30 where classifiers of the *device* category are declared for the light sensor, the sonar, the wheel assemblies and the gripper, which is used to lift the carried object. Note that such gripper does not actually exist for the Mindstorm kit but is added for illustration purposes only. The component types for the sensors and actuators declare interaction features whose types for the corresponding physical resources are provided by the *Physics* package of Listing 18. They also declare bus access features typed with classifiers for the sensor and actuator buses. Those will represent the standard RJ12 connectors provided with the NXT kit. Note that no component implementation is provided for sensors and actuators since their internal composition is irrelevant for this modelling.

```
device Light_Sensor extends Physics::Power_Consuming_Object
    features
        light_in: in feature Physics::Light;
        light_data: requires bus access Sensor_Bus;
    properties
        Physics_Properties::Mass => 20.0 g;
end Light_Sensor;

device Sonar extends Physics::Power_Consuming_Object
    features
        sound_in: in feature Physics::Sound;
        sound_out: out feature Physics::Sound;
        sound_data: requires bus access Sensor_Bus;
    properties
        Physics_Properties::Mass => 20.0 g;
```

```
end Sonar;

device Wheel_Assembly extends Physics::Power_Consuming_Object
    features
        power_in: refined to requires bus access Actuator_Bus;
        force: out feature Physics::Force;
    properties
        Physics_Properties::Mass => 100.0 g;
end Wheel_Assembly;

device Gripper extends Physics::Power_Consuming_Object
    features
        power_in: refined to requires bus access Actuator_Bus;
        force: out feature Physics::Force;
    properties
        Physics_Properties::Mass => 20.0 g;
end Gripper;
```

Listing 30: Classifiers for robot sensors and actuators

Now that classifiers have been declared for all the components of the NXT Mindstorm robot kit, they can be instantiated and assembled into a robot plant system configured for object transportation in the warehouse. This is illustrated by Listing 31 and its corresponding diagram of figure 8.17. New *Line_Follower_Robot* classifiers are provided extending the partially-configured basic robot classifiers previously defined, which only contains a brick subcomponent (Listing 27 line 10). The extending *Line_Follower_Robot* system type adds features specific to the line following configuration for using the wheel assemblies, the gripper and the light and sonar sensors (line 2).

```
system Line_Follower_Robot extends Robots_Library::Robot
    features
        light_sensor_in: in feature Physics::Light;
        sonar_in: in feature Physics::Sound;
        sonar_out: out feature Physics::Sound;
        force_left_wheel: out feature Physics::Force;
        force_right_wheel: out feature Physics::Force;
        force_gripper: out feature Physics::Force;
end Line_Follower_Robot;

system implementation Line_Follower_Robot.nxt extends Robots_Library::
   Robot.basic
    subcomponents
        left_wheel_assembly: device Robots_Library::Wheel_Assembly;
        left_wheel_cable: bus Robots_Library::Actuator_Bus;
        righ_wheel_assembly: device Robots_Library::Wheel_Assembly;
        rght_wheel_cable: bus Robots_Library::Actuator_Bus;
        light_sensor: device Robots_Library::Light_Sensor;
        light_sensor_cable: bus Robots_Library::Sensor_Bus;
        sonar: device Robots_Library::Sonar;
        sonar_cable: bus Robots_Library::Sensor_Bus;
        gripper: device Robots_Library::Gripper;
        gripper_cable: bus Robots_Library::Actuator_Bus;
        brick: refined to system Robots_Library::NXT_Brick.basic;
    connections
        brick_left_wheel_cable: bus access brick.out_1 ->
           left_wheel_cable;
        left_wheel_cable_left_wheel_assembly: bus access left_wheel_cable
            -> left_wheel_assembly.power_in;
```

```
        left_wheel_force_robot: feature left_wheel_assembly.force ->
            force_left_wheel;
        brick_right_wheel_cable: bus access brick.out_2 ->
            rght_wheel_cable;
        right_wheel_cable_right_wheel_assembly: bus access
            rght_wheel_cable -> righ_wheel_assembly.power_in;
        right_wheel_force_robot: feature righ_wheel_assembly.force ->
            force_right_wheel;
        brick_gripper_cable: bus access brick.out_3 -> gripper_cable;
        gripper_cable_gripper: bus access gripper_cable -> gripper.
            power_in;
        gripper_force_robot: feature gripper.force -> force_gripper;
        robot_light_sensor: feature light_sensor_in -> light_sensor.
            light_in;
        light_sensor_light_sensor_cable: bus access light_sensor.
            light_data -> light_sensor_cable;
        light_sensor_cable_brick: bus access light_sensor_cable -> brick.
            in_1;
        sonar_out_robot: feature sonar.sound_out -> sonar_out;
        robot_sonar_in: feature sonar_in -> sonar.sound_in;
        sonar_sonar_cable: bus access sonar.sound_data -> sonar_cable;
        sonar_cable_brick: bus access sonar_cable -> brick.in_2;
end Line_Follower_Robot.nxt;
```

Listing 31: Classifiers for the line follower robot plant

The configured robot system implementation (line 11) instantiates sensor and actuator subcomponents including buses for the RJ12 connectors. The generic brick subcomponent (line 23) of the extended generic robot is refined to the NXT brick of Listing 28 (line 12) and figure 8.16.

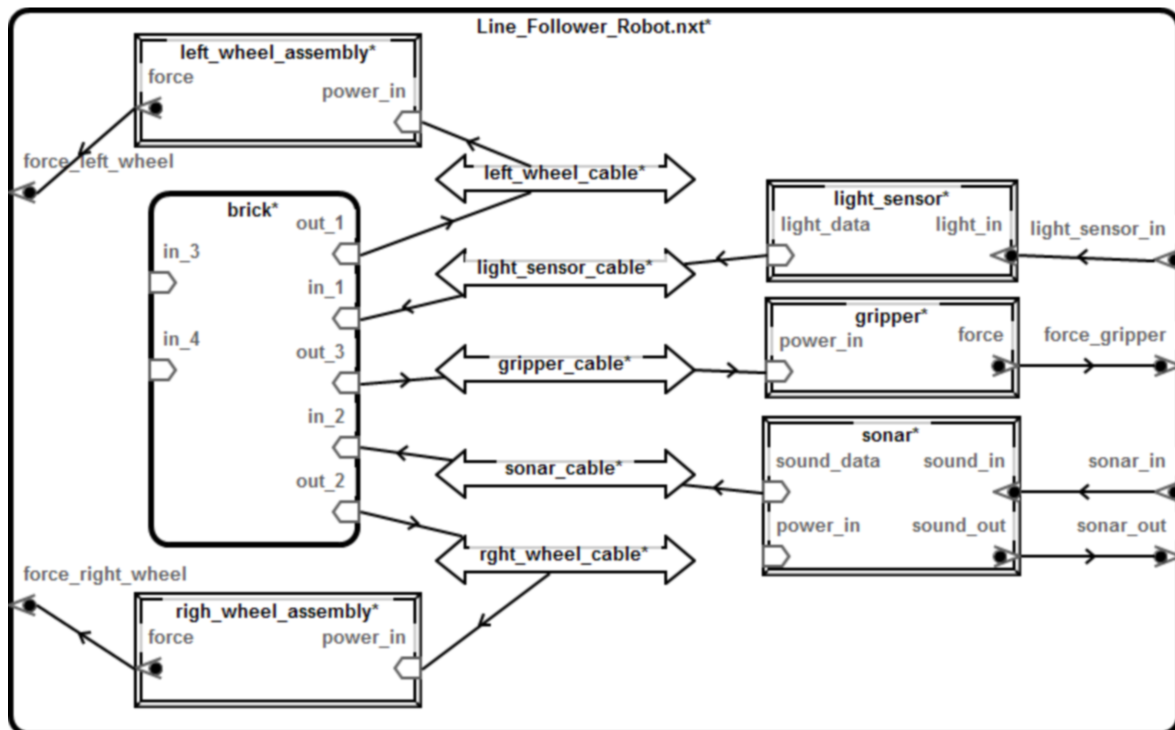


Fig. 8.17: The plant model for the Lego NXT line follower robot

This constitutes the physical plant model of our robot carrier system. Such hardware plant will be used to deploy a software application (the cyber part of the robot CPS) to control the speed of the wheels according to the data sensed by the light and sonar sensors.

8.6.6 Deployment

This section considers concrete implementations in software of the system functions introduced in section 8.6.4 and the deployment of the software onto the plant model introduced in section 8.6.5: the NXT Mindstorm Lego robot configured for carrying objects in the warehouse. In order to deploy the system, the system functions first need to be refined into a software application executable by the NXT brick of the plant model.

Software Application

Constructs

Software applications can be modelled in AADL using the six software component sub-categories introduced below:

- *Data*: A data component represents hierarchical data structures such as instance variables of a class in object-oriented programming languages or the fields of a record. Operations on a data type can also be modelled by declaring a *provides subprogram access* feature on the data component type. The feature is typed with classifiers of the accessed subprogram.
- *Subprogram*: A subprogram represents an atomic operation with parameters. It cannot have any state persisting after the call (static data), but can have local variables represented by data subcomponents declared in the subprogram implementation. Subprogram parameters and required access to data must be explicitly declared in a subprogram type. Events raised within a subprogram can also be specified as event or event data ports in a subprogram type.
- *Subprogram group*: A subprogram group represents subprogram libraries, whose content is declared through a subprogram group type. Subprogram groups are made accessible to other components using *subprogram group access* features that can be connected with *subprogram group access connections*. This allows for reducing the number of connections by providing a single connection for the whole group.
- *Thread*: A thread represents a scheduleable unit that can be executed concurrently with other threads. It represents sequential flows of control executing instructions. Threads can be dispatched periodically or upon the arrival of data or events on ports, or upon arrival of subprogram calls from other threads.
- *Thread group*: A thread group represents logically grouped threads. A thread group type specifies the features and required subcomponent accesses for thread subcomponents declared in a thread group implementation also declaring the connections between threads.
- *Process*: A process represents a virtual memory address space. This address space contains the program formed by the process's subcomponents, which can be threads, thread groups, subprogram, subprogram group or data subcomponents.

The AADL standard includes several annexes supporting software modelling:

- The Behaviour Annex (BA) consists of a state-machine based sublanguage allowing modelling the behaviour of AADL components. From this neutral behavioural description, executable code can be generated. However presenting the BA is beyond the scope of this chapter.
- The data modelling annex, which consists of predefined properties and classifiers provides guidance to represent data types and structures in AADL to be used for code generation.
- The ARINC653 annex, which consists of predefined properties and classifiers, provides a standard way of representing ARINC653 compliant partitioned embedded system architectures in AADL. Again, this annex is beyond the scope of this chapter.
- The code generation annex defines language-specific rules for generation of runtime systems from AADL models.

The refined software model presented in this section will provide subprogram classifiers extending the system function abstract classifiers. Thread classifiers will be provided for specifying scheduling properties for these software subprograms. Software-specific properties for supporting timing and scheduling analyses and code generation will be set to these components.

The execution semantics of a thread is specified using predefined properties, which are found under the standard *Thread_Properties* property set. Those properties are:

- *Dispatch_Protocol*, which can have the values of:
  - *Periodic*: the thread is activated periodically
  - *Aperiodic*: the thread is activated upon receiving messages
  - *Sporadic*: the thread is activated upon receiving messages with a minimum delay between two activations
  - *Timed*: the thread is activated *either* upon receiving a message or a given deadline (the timer is reset upon message reception)
  - *Hybrid*: the thread is activated *both* upon receiving a message and a given deadline
- *Scheduling_Protocol*: specifies the scheduling policy
- *Compute_Execution_Time*: represents the execution time of a thread or a subprogram
- *Stack_Size*: Defines the size of the stack for the thread

Threads can communicate with each other using three predefined communication mechanisms represented by their port and connection types:

- *Data ports* represent unenqueued state data
- *Event data ports* represent queued message data
- *Event ports* represent asynchronous events

Port connections can have different timing semantics:

- *sampled*: the receiving thread samples at dispatch or during execution
- *immediate*: the data is communicated upon execution completion of the sending thread
- *delayed*: the data is communicated upon deadline of the sending thread

The standard *Programming_Prperties* property set defines properties for code generation:

- *Source_Language* specifies the programming language. Predefined values are Ada95, Ada2005, C, Java, Simulink_6_5 and SCADE
- *Source_Text* specifies a source code file name
- *Source_Name* specifies the name of a data structure or function in the code file specified by *Source_Text*.
- *Compute_Entrypoint_Call_Sequence* specifies the name of a call sequence in a thread that will execute after the thread has been dispatched.

Running Example Specification

A package is created to contain declarations for the software classifiers as shown in Listing 32. Concrete data types are declared for the software variables, which are derived from the system function variables by extending them and adding a *SW* suffix to their names. To these classifiers, properties specifying their concrete representation in code are set using the *Data_Representation* property of the data model annex (line 7). The *Robot_State_SW* software variable is also refined by providing properties for an equivalent data structure *Robot_state* declared in a C code file *data_types.h* and to be used for code generation (lines 24 and 25).

```
package Line_Follower_Software
public
    with Line_Follower_Functions, Code_Generation_Properties, Base_Types,
        Data_Model;

    data Light_Intensity_SW extends Physics::Light
        properties
```

```
            Data_Model::Data_Representation => integer;
            Data_Size => 4 bytes;
    end Light_Intensity_SW;

    data Turn_Angle_SW extends Line_Follower_Functions::Turn_Angle
        properties
            Data_Model::Data_Representation => integer;
    end Turn_Angle_SW;

    data Power_SW extends Line_Follower_Functions::Power
        properties
            Data_Model::Data_Representation => integer;
    end Power_SW;

    data Robot_State_SW extends Line_Follower_Functions::Robot_State
        properties
            Data_Model::Data_Representation => Enum;
            Data_Model::Enumerators => ( "FORWARD", "STOP" );
            Source_Name => "Robot_state";
            Source_Text => ("data_types.h");
    end Robot_State_SW;
```

Listing 32: Data classifiers for the software application

Concrete software subprogram implementations for the *compute turn angle* and the *compute wheels motor power* functions are declared as shown in Listing 33. Like for the data types, those subprograms extend the system function abstract components and refine the abstract features to subprogram parameters typed with appropriate SW data types previously introduced.

```
subprogram Compute_Turn_Angle_SW extends Line_Follower_Functions::
   Compute_Turn_Angle
    features
        light_intensity: refined to in parameter Light_Intensity_SW;
        turn_angle:  refined to out parameter Turn_Angle_SW;
    properties
        Classifier_Substitution_Rule => Type_Extension;
        Source_Language => (C);
end Compute_Turn_Angle_SW;

subprogram implementation Compute_Turn_Angle_SW.pid extends
   Line_Follower_Functions::Compute_Turn_Angle.pid
    properties
        Source_Name => "computePID";
        Source_Text => ("line_follower.h", "line_follower.c");
end Compute_Turn_Angle_SW.pid;

subprogram Compute_Wheels_Motors_Power_SW extends Line_Follower_Functions
   ::Compute_Wheels_Motors_Power
    features
        turn_angle: refined to in parameter Turn_Angle_SW;
        left_motor_power: refined to out parameter Power_SW;
        right_motor_power: refined to out parameter Power_SW;
        state: refined to requires data access Robot_State_SW;
    properties
        Classifier_Substitution_Rule => Type_Extension;
        Source_Language => (C);
end Compute_Wheels_Motors_Power_SW;
```

```
subprogram implementation Compute_Wheels_Motors_Power_SW.basic extends
   Line_Follower_Functions::Compute_Wheels_Motors_Power.basic
    properties
        Source_Name => "computeWheelsPower";
        Source_Text => ("line_follower.h", "line_follower.c");
end Compute_Wheels_Motors_Power_SW.basic;
```

Listing 33: Subprogram classifiers for the compute turn angle and the compute wheels motor power software functions

The implementation of the *compute_turn_angle* subprogram makes use of a PID control mechanism, which computes a turn angle that is proportional to the observed light intensity as illustrated on the RHS of figure 8.8. The C code for such subprogram can actually be derived from control simulations with tools such as Matlab Simulink. This code can then be associated with an AADL subprogram classifier via the *Source_Language*, *Source_Text* and *Source_Name* properties as shown in Listing 33 (lines 7, 12 and 13). Note that the *Source_Language* property is set on the subprogram types (lines 7 and 24) while the other source code properties are set on the subprogram implementations. This ease design space exploration by capturing the variability of different subprogram code implementations of a given programming language.

In order to read the value from the light sensor and to actuate the wheel motors, subprograms provided by the NXT OSEK ECRobot C library [219] are used. A new subprogram implementation is created to encapsulate calls to the *get_light_intensity*, *follow_line*, *set_left_motor_power* and *set_right_motor_power* subprograms as shown in Listing 34 and in figure 8.18. Implementation details are added consisting of data subcomponents for setting the value of required input parameters for the ECRobot subprograms such as the port number and a braking coefficient and initial values on data subcomponents (lines 8 to 19) for the corresponding data structures in code.

```
subprogram Follow_Line_SW_NXT
    features
        state: requires data access Robot_State_SW;
end Follow_Line_SW_NXT;

subprogram implementation Follow_Line_SW_NXT.basic
    subcomponents
        left_wheel_port: data Base_Types::Integer {
            Data_Model::Initial_Value => ( "NXT_PORT_B" );
        };
        left_wheel_brake: data Base_Types::Integer {
            Data_Model::Initial_Value => ( "0" );
        };
        right_wheel_port: data Base_Types::Integer {
            Data_Model::Initial_Value => ( "NXT_PORT_A" );
        };
        right_wheel_brake: data Base_Types::Integer {
            Data_Model::Initial_Value => ( "0" );
        };
    calls
        main_call: {
            get_light_intensity: subprogram ECRobot_Get_Light_Intensity;
            follow_line: subprogram Follow_Line_SW.basic;
            set_left_motor_power: subprogram NXT_Motor_Set_Power;
            set_right_motor_power: subprogram NXT_Motor_Set_Power;
        };
    connections
        light_intensity_follow_line: parameter get_light_intensity.
           light_intensity -> follow_line.light_intensity;
```

```
        left_motor_power: parameter follow_line.left_motor_power ->
            set_left_motor_power.power;
        right_motor_power: parameter follow_line.right_motor_power ->
            set_right_motor_power.power;
        state_follow_line: data access state -> follow_line.state;
        left_wheel_port_set_left_motor_power: parameter left_wheel_port
            -> set_left_motor_power.portNb;
        left_wheel_brake_set_left_motor_power: parameter left_wheel_brake
             -> set_left_motor_power.brake;
        right_wheel_port_set_right_motor_power: parameter
            right_wheel_port -> set_right_motor_power.portNb;
        right_wheel_brake_set_right_motor_power: parameter
            right_wheel_brake -> set_right_motor_power.brake;
    properties
        Compute_Execution_Time => 3 ms .. 5 ms;
end Follow_Line_SW_NXT.basic;
```

Listing 34: Subprogram classifiers for the follow line software function

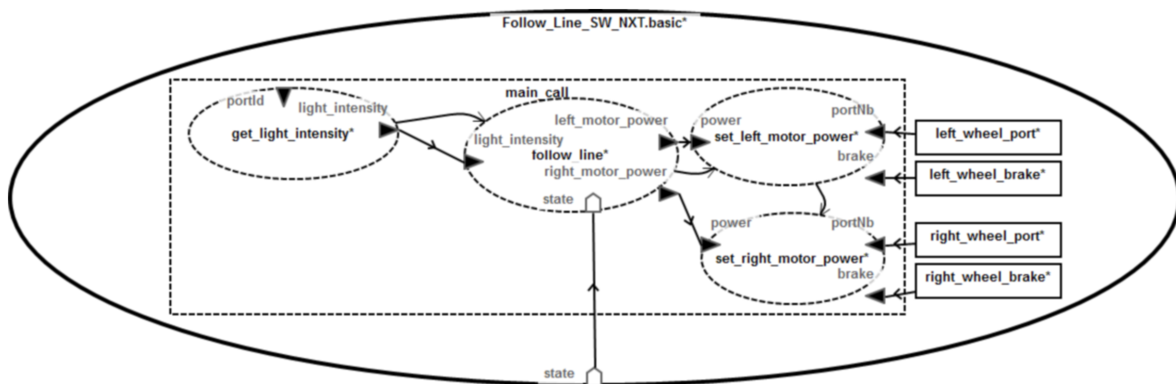


Fig. 8.18: The NXT line following subprogram

Software subprograms extending the *Pick_Up_Object* and *Drop_Off_Object* system function components must also be declared. Those are not presented here but can be found in the complete set of models from [63].

The next step is to specify how the subprograms of the software application will be executed. The *Pick_Up_Object_SW.basic* subprogram should be executed once the robot has reached the start of the path. The *Follow_Line_SW_NXT.basic* subprogram should be executed once the object has been picked up and periodically at a frequency high enough to follow the line, given a required tolerance, the minimum line curvature radius and the robot speed. The *Drop_Off_Object_SW.basic* subprogram should be executed once the end of the path is reached. The *Detect_Obstacles_SW* subprogram should be executed periodically while the line is being followed at a frequency high enough to detect obstacle on time in order to avoid collisions. Finally, the *Logging_SW* subprogram should be always executed periodically, with a period given by logging requirements.

The modelling of these threads is shown in Listing 35. The *Pick_Up_Object_Thread* thread is set with an *at_beginning_path* input event port for its activation when the robot reaches the beginning of the path and an *object_picked* output event port for notification when the object has been picked by the gripper. An *aperiodic* dispatch protocol is set meaning that the thread will be dispatched upon events on its input event port.

```
thread Pick_Up_Object_Thread
    features
        at_beginning_path: in event port;
        object_picked: out event port;
    properties
        Dispatch_Protocol => aperiodic;
```

```
        Deadline => 5 ms;
        Priority => 2;
        Stack_Size => 512 Bytes;
end Pick_Up_Object_Thread;

thread Follow_Line_Thread
    features
        state: requires data access Robot_State_SW;
        object_picked: in event port;
        object_dropped: in event port;
        at_beginning_path: out event port;
        at_end_path: out event port;
        right_wheel: out data port Power_SW;
        left_wheel: out data port Power_SW;
    flows
        light_to_left: flow path light_intensity -> left_wheel;
        light_to_right: flow path light_intensity -> right_wheel;
    properties
        Dispatch_Protocol => periodic;
        Period => 15 ms;
        Deadline => 15 ms;
        Priority => 3;
        Stack_Size => 512 Bytes;
end Follow_Line_Thread;

thread implementation Follow_Line_Thread.basic
    calls
        main_call: {
             follow_line: subprogram Follow_Line_SW_NXT.basic;
        };
    connections
        state_follow_line: data access state -> follow_line.state;
    properties
        Compute_Entrypoint_Call_Sequence => reference (main_call);
        Latency => 3 ms .. 5 ms applies to light_to_left, light_to_right;
end Follow_Line_Thread.basic;

thread Detect_Obstacles_Thread
    features
        state: requires data access Robot_State_SW;
    properties
        Dispatch_Protocol => periodic;
        Period => 10 ms;
        Deadline => 5 ms;
        Priority => 2;
        Stack_Size => 512 Bytes;
end Detect_Obstacles_Thread;

thread Log_Thread
    properties
        Dispatch_Protocol => periodic;
        Period => 20 ms;
        Deadline => 5 ms;
        Priority => 5;
        Stack_Size => 512 Bytes;
```

```
end Log_Thread;
```

Listing 35: Thread classifiers for the tasks of the software application

The *Follow_Line_Thread* thread is set with an *object_picked* input event port for its activation when the object has been picked by the gripper and an *object_arrived* output event port for its deactivation and a periodic dispatch protocol. The determination of the period of this thread will be detailed in the analysis section 8.6.7. A thread implementation is provided specifying a call to the follow line subprogram (line 21) and a data access connection for the robot state variable, which will be valued by the obstacle detection subprogram. Finally, the *Compute_Entrypoint_Call_Sequence* property (line 35) specifies which call sequence (there could be many) should be executed upon dispatch of the thread.

A *Drop_Off_Object_Thread* thread classifier (not shown in the listing due to space constraints) is provided for executing the *Drop_Off_Object_SW* subprogram with features and properties similar to the ones of the *Pick_Up_Object_Thread* thread. Another thread classifier is provided for the obstacle detection subprogram with a data access feature for the robot state and a periodic dispatch protocol. Finally, the logging function is encapsulated within a periodic thread. More details on other properties of these threads can be found in the analysis section 8.6.7.

Now that these thread classifiers have been declared, they can be instantiated as subcomponents of a *process* implementation, which represents a memory address space for the global software application. This is shown in Listing 36 and figure 8.19, where a data subcomponent is declared for the robot state data with a *FORWARD* initial value. Proper port connections are declared between the subcomponents.

```
process Carry_Object_Process
    features
        light_intensity: in data port Light_Intensity_SW;
        left_wheel_power: out data port Power_SW;
        right_wheel_power: out data port Power_SW;
    flows
        light_to_left: flow path light_intensity -> left_wheel_power;
        light_to_right: flow path light_intensity -> right_wheel_power;
end Carry_Object_Process;

process implementation Carry_Object_Process.basic
    subcomponents
        state: data Robot_State_SW {
            Data_Model::Initial_Value => ( "FORWARD" );
        };
        pick_up_object: thread Pick_Up_Object_Thread;
        follow_line: thread Follow_Line_Thread.basic;
        drop_off_object: thread Drop_Off_Object_Thread;
        detect_obstacle: thread Detect_Obstacles_Thread;
        logging: thread Log_Thread;
    connections
        pick_up_object_folow_line: port pick_up_object.object_picked ->
           follow_line.object_picked;
        follow_line_drop_object: port follow_line.at_end_path ->
           drop_off_object.object_arrived;
        drop_object_follow_line: port drop_off_object.object_dropped ->
           follow_line.object_dropped;
        obstacle_detection_state: data access detect_obstacle.state ->
           state;
        state_follow_line: data access state -> follow_line.state;
        follow_line_begin_path_pick_up_object: port follow_line.
           at_beginning_path -> pick_up_object.at_beginning_path;
        light_intensity_follow_line: port light_intensity -> follow_line.
           light_intensity;
```

```
        follow_line_right_wheel: port follow_line.right_wheel ->
           right_wheel_power;
        follow_line_left_wheel: port follow_line.left_wheel ->
           left_wheel_power;
    properties
        Timing => Immediate applies to follow_line_right_wheel,
           follow_line_left_wheel;
end Carry_Object_Process.basic;
```

Listing 36: Process classifiers for the software application

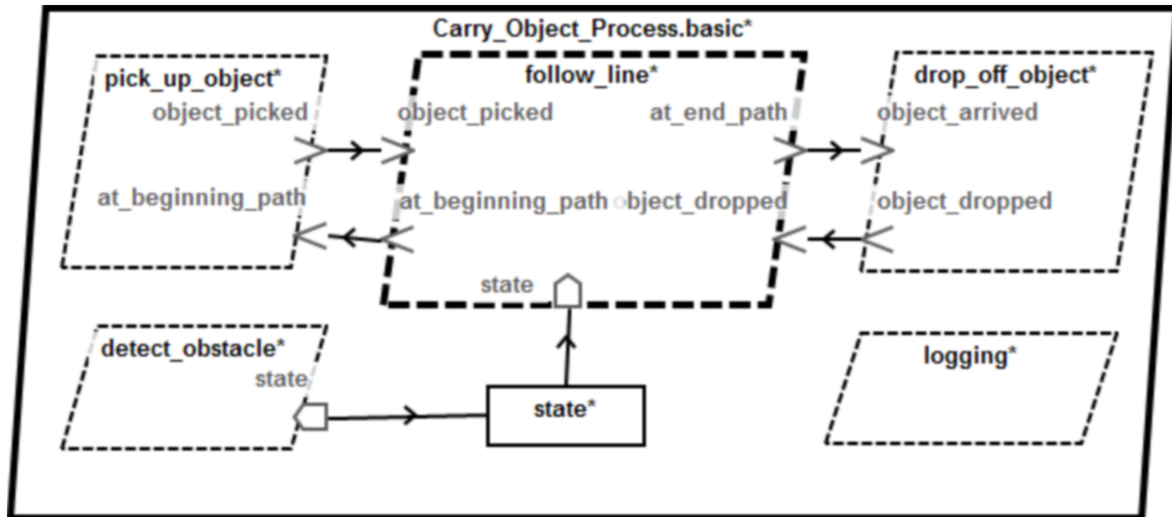


Fig. 8.19: The carry object software process implementation

This constitutes a complete software application that can be deployed on the brick of the NXT robot plant model. However, as will be presented in the analysis section, we are interested in determining the latency of steering the robot upon a change in the value of the observed path light intensity. For this purpose, we extend the software model by modeling the drivers of the input and output circuits of the brick driving the light sensor and the two motor assemblies.

In order to achieve this, we first create a system component for the software application as illustrated in figure 8.20. To this system we add a subcomponent of the software application process previously defined and device subcomponents with data ports typed with the software power and light intensity variable types. Such data ports are connected to the software process containing the threads. As explained in section 8.6.5, devices can be used for both hardware and software parts of systems. For software, they represent the software part of corresponding hardware devices, such as drivers residing in a memory and executed on an external processor. Such is the case for the input and output circuit devices of the robot brick (figure 8.16), which include built-in drivers to control sensors and actuators.

Binding of the Software Application on the Execution Platform

The last step in modeling the robot CPS is to deploy the software application on the plat model. This is presented in the following.

Concepts

In order to specify the binding of the software application on the plant model, first a system implementation must be provided for the *Line_Follower_Robot_CPS*, for the system type of the system overview of Listing 19. This implementation should have as subcomponent the software application previously defined and the robot plant model.

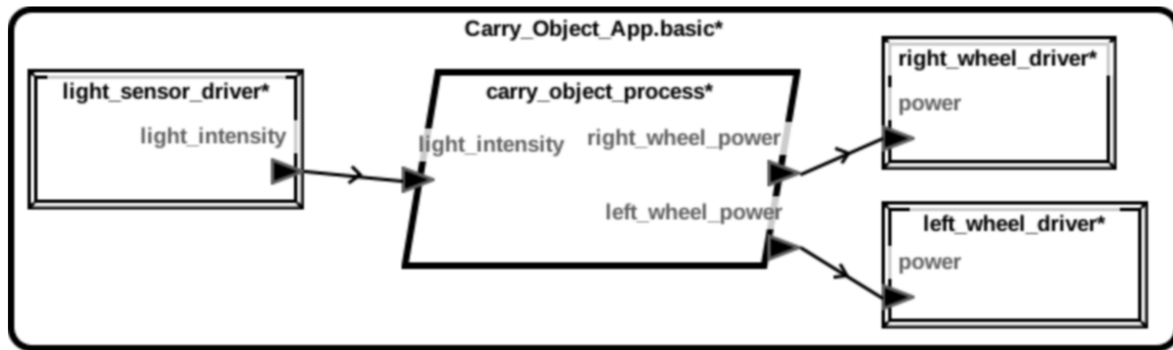


Fig. 8.20: A system component for the software application including devices for modeling drivers of sensors and actuators

The binding of the thread and data subcomponents of the process on the execution platform components can then be specified. Properties from the standard *Deployment_Properties* property set are used for that:

- *Actual_Processor_Binding* specifies the processor that will execute threads or the hardware device that will execute software devices (driver)
- *Actual_Memory_Binding* specifies the memory that will store processes and data.
- *Actual_Connection_Binding* specifies the bus that will transmit the data of port connections.

Running Example Specification

The system implementation for the line follower CPS component type is shown in Listing 37 and figure 8.21. A system subcomponent for the NXT software application and a system subcomponent for the configured NXT line following robot plant model are instantiated. The features of the robot plant model are connected to the features of the CPS system type of the system overview (Listing 19).

```
system implementation Line_Follower_Robot_Cps.nxt
    subcomponents
        app: system Line_Follower_Software::Carry_Object_App.basic;
        robot: system Line_Follower_Robot.nxt;
    connections
        force_left_wheel_conn: feature robot.force_left_wheel ->
           force_left_wheel;
        force_right_wheel_conn: feature robot.force_right_wheel ->
           force_right_wheel;
        light_sensor_conn: feature light_sensor_in -> robot.
           light_sensor_in;
        force_gripper_conn: feature robot.force_gripper -> force_gripper;
        sonar_in_conn: feature sonar_in -> robot.sonar_in;
        sonar_out_conn: feature robot.sonar_out -> sonar_out;
    properties
        Actual_Processor_Binding => (reference (robot.brick.
           main_processor)) applies to app.carry_object_process;
        Actual_Memory_Binding => (reference (robot.brick.main_processor.
           ram)) applies to app.carry_object_process;
        Actual_Processor_Binding => (reference (robot.brick.input_circuit
           )) applies to app.light_sensor_driver;
        Actual_Connection_Binding => (reference (robot.brick.data_bus))
           applies to app.light_sensor_process_conn;
        Actual_Processor_Binding => (reference (robot.brick.
           output_circuit)) applies to app.left_wheel_driver;
```

```
        Actual_Connection_Binding => (reference (robot.brick.data_bus))
            applies to app.process_left_wheel_conn;
        Actual_Processor_Binding => (reference (robot.brick.
            output_circuit)) applies to app.right_wheel_driver;
        Actual_Connection_Binding => (reference (robot.brick.data_bus))
            applies to app.process_right_wheel_conn;
end Line_Follower_Robot_Cps.nxt;
```

Listing 37: System implementation for the Robot CPS

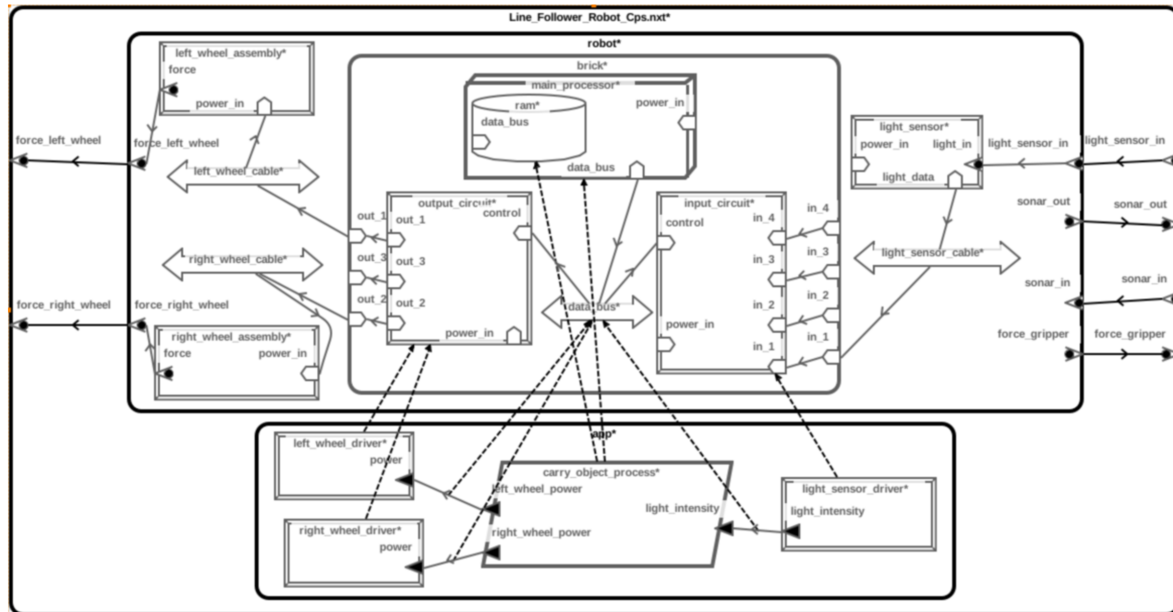


Fig. 8.21: The deployed line follower robot CPS

The binding properties *Actual_Processor_Binding* and *Actual_Memory_Binding* are set to respectively the ARM7 processor and its contained memory as shown in Listing 37 under the properties section (lines 13 and 14). This is also illustrated in figure 8.21 with the dashed arrows pointing from the process subcomponent to the NXT brick ARM processor and its included RAM memory. Since there is only one processor and one memory, it was chosen to apply the binding properties to the enclosing process, which means that all contained threads and data subcomponents are bound to the same processor and memory. If several processors were available, a binding property could have been be set for each individual thread for specifying their execution by different processors.

The device drivers for the light sensor and the wheel motors are also bound to the input and output circuits on the NXT brick. The port connections between the devices and the process are bound to the data bus connecting the input and output circuits to the ARM processor.

8.6.7 Analyses

The modelling of the object carrier robot CPS has been completed. The next step is to verify some of its properties to ensure that the system will operate properly and will meet its requirements. Furthermore, it would be interesting to estimate its performances. A first question to ask is if the ARM 7 processor can schedule all the threads given their timing properties. This strongly depends on the computation time of the subprograms themselves. Several methods exist to answer this question. For the case of the NXT brick, assume that the times are given by executing the programs alone on the brick and by logging the times to the display of the robot so that they can be memorized.

Scheduling

Scheduling analysis can be performed from the software application of section 8.6.6 and using tools such as Cheddar or its commercial implementation in AADL Inspector. This analysis will rely on the following properties attached to software components:

In Listing 35, property values are given to AADL threads in order to define their dispatch protocol, period, priority, and their sequence of subprogram calls. From the call sequence of a thread, we can compute an upper bound on the thread execution time by summing the upper bound of the called subprograms execution time. The latter is given for subprogram *Follow_Line_SW_NXT* in listing 34 using the *Compute_Execution_Time* property. Last but not least, the scheduling protocol is attached to the AADL processor component on which threads are going to run. This is done in Listing 28 using the *Scheduling_Protocol* property (line 34). Here, the scheduling protocol is a well known protocol named Rate Monotonic Scheduling (RMS). RMS is a fixed priority scheduling technique (tasks are executed according to a predefined priority) and tasks priorities are ordered as follows: the most frequent tasks are given a higher priority.

Semantics

The AADL modeling elements listed above define a periodic task set model (where an AADL thread is a periodic task) which can be summarized on figure 8.22.

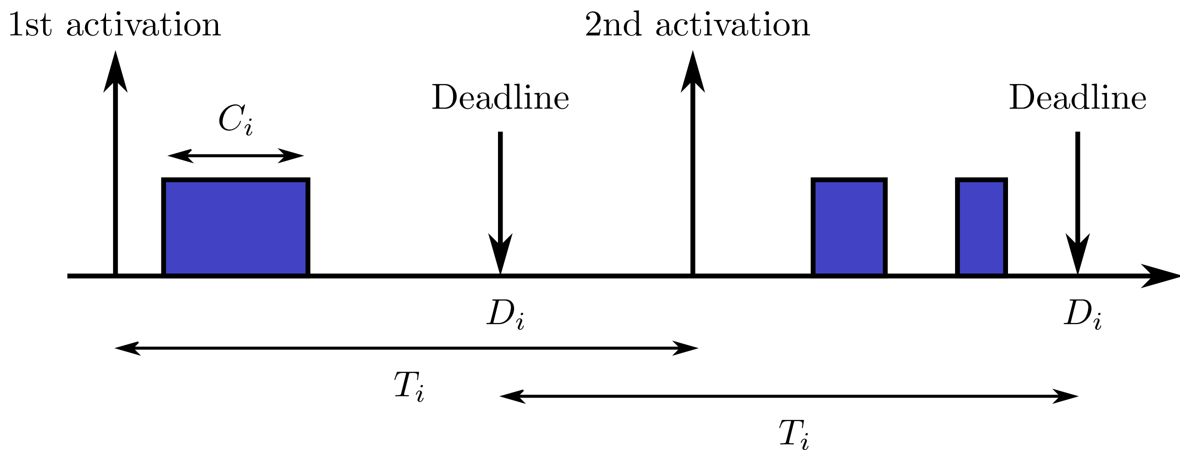


Fig. 8.22: Periodic task set

On this figure, a task τ_i is characterized by its period T_i, its deadline D_i, its capacity C_i. The capacity of a task corresponds to the upper bound on its execution time, and we explained above how this can be extracted from the execution time of AADL subprograms called in sequence by an AADL thread. In practice, we measure the execution time of subrpograms running on the NXT brick to estimate tasks capacity.

From this task model, the CPU utilization of U_i of a task τ_i can be computed as follows: $U_i = \frac{C_i}{T_i}$ and the CPU utilization of the task is simply the sum of every task utilization.

Given a periodic task set model, schedulability analysis can be performed to ensure all the tasks of the object carrier robot can finish their execution within their deadlines. A first option to perform this analysis is to compare the worst-case processor utilization with a safe upper bound: it has been proven by Liu and Layland in 1973 that a necessary condition for the schedulability of a task set scheduled with RMS, on a mono-core processor, is that the CPU utilisation of the task set remains under 69.3147%. Another option, for task sets scheduled with a fixed priority scheduling, is to compute tasks Worst Case Response Time (WCRT) and check that they remain inferior to the tasks deadline. The WCRT of a task τ_i, on a mono-core processor, can be computed using the following recursive formula:

$$WCRT_i = \sum_{\tau_j \in HP_i} \lceil \frac{WCRT_i}{T_j} \rceil \cdot C_j$$

where HP_i is the set of tasks having a higher priority than τ_i.

Analysis results

Figures 8.23 and 8.24 show the different results provided by AADL Inspector after performing the scheduling analysis of an AADL model.

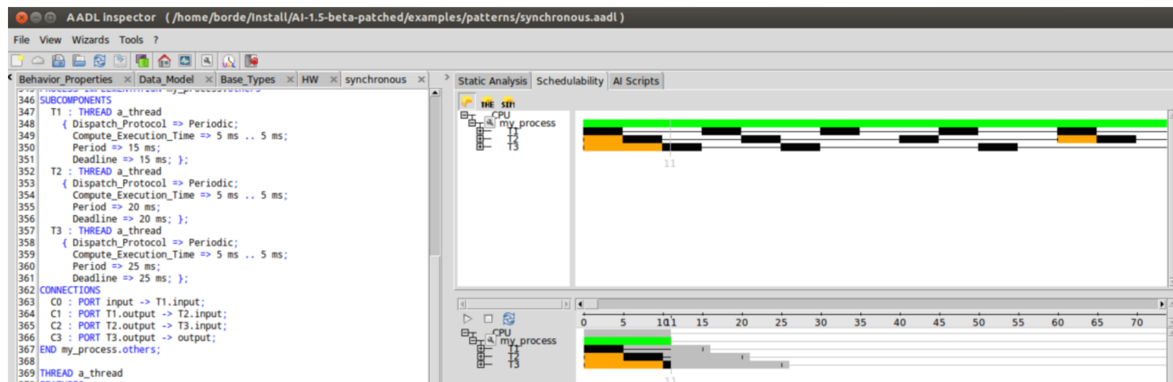
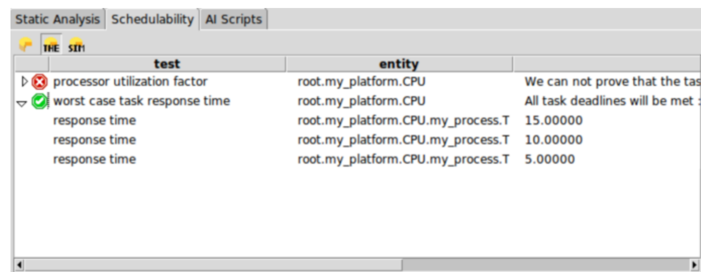


Fig. 8.23: Tasks execution gantt chart



test	entity	
processor utilization factor	root.my_platform.CPU	We can not prove that the tas
worst case task response time	root.my_platform.CPU	All task deadlines will be met :
response time	root.my_platform.CPU.my_process.T	15.00000
response time	root.my_platform.CPU.my_process.T	10.00000
response time	root.my_platform.CPU.my_process.T	5.00000

Fig. 8.24: Response time analysis result

This analysis can help us configure the task set either by modifying the period of AADL threads, reducing the number of tasks executed on the platform, or changing the execution platform itself. In the case of the object carrier robot, given the number of tasks, their execution time, and the physics of the system, the scheduling analysis helped to define tasks period.

Latency

Another question to ask about our object carrying robot is whether it will be able to follow the line given a minimum speed requirement. For this, the latency of the system in steering the robot must be known. In particular, we are interested in the latency of the data flows involved in the control loop, i.e. the amount of time separating the production of a data from the light sensor, to the production of commands on the engine. AADL is well suited for such analysis using its *flow* constructs.

Constructs

AADL flows are used to annotate AADL elements in order to specify paths of information flow across components. As such, flows do not represent any concrete system architecture elements but facilitate analyses and allow characterizing information flows with additional properties. By defining connections among directed ports, possible data flows are already defined by the structure of an AADL model. However, they are **possible**

data flows and potentially some of them are not present in the software application. Besides, only a subset of data flows require latency analysis.

In AADL, a flow consists of:

- *Flow source*: describes the origin of an information flow
- *Flow path*: describes a path of information flow between components
- *Flow sink*: describes the end of an information flow
- *End to end flow*: describes a complete flow including a source, a path and a sink

Component types can only contain flow specifications (sources, sinks and paths) annotating the declared features.

Component implementations can only contain flow implementations (sources, sinks and paths) annotating features, connections and subcomponent flows. They can also contain end to end flows beginning by a flow source and ending by a flow sink.

Latency upper bounds can either be attached to end-to-end flows or set in ReqSpec requirements assigned to end-to-end flows. Latency contributions are attached to model elements implementing these flows. In practice, the latency of an end-to-end flow is the sum of latencies contributed by components and connections traversed by the flow.

Figure 8.25 represents the architecture of the robot carrier application along with devices representing the light sensor and the right and left motors. They are represented with an AADL device, representing the logical execution of the device driver. In addition, the end-to-end flow from the light sensor output (flow source) to the left motor input (flow sink) passing by the software application (flow path in *carry_object_process*) is coloured in yellow.

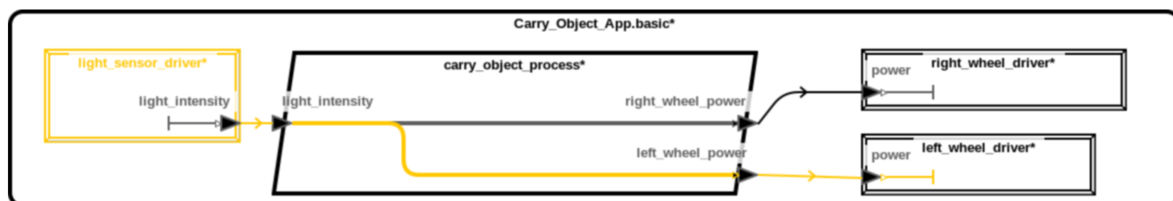


Fig. 8.25: Light sensor to left motor end-to-end flow

There exists three types of contributions to a flow latency in AADL:

1. Latency property value can be attached to a flow path in order to decompose the end-to-end latency into latencies attached to subcomponents.
2. Physical communications (i.e. connections bound to a bus component) contribute to the latency of flows passing by these links.
3. Logical communications (i.e. communications among software components) also contribute to the latency of flows passing by these links.

We explain further how to use these modeling capabilities in the remainder of the following paragraph, illustrating these types.

Running Example Specification

Contribution type 1: Latency property value attached to a flow path. In the early phase of the development process, or in order to decompose the end-to-end latency requirement into latency requirements attached to subcomponents, AADL users can attach flow latency values to flow path as illustrated in Listing 35 (property *Latency* defined in *Follow_Line_Thread.basic*). Note that without this property, the deadline of the thread would be used as an upper bound (and 0 as a lower bound).

Contribution type 2: Physical communications contribution. When a flow passes by a logical connection (i.e. connection between software components) bound to a bus component, the communication itself takes time. AADL offers the possibility to model this contribution as a linear function of the number of transmitted bytes.

Property *Transmission_Time* is meant for this usage, which is illustrated in Listing 38. In this listing we can see that the *Transmission_Time* property consists of two fields: *Fixed* and *PerByte* which respectively represent a communication offset and a communication time per byte. Each of these fields is a time range, representing the lower and upper bound of the communication offset and time per byte. Note that when this property is used, it is necessary to define the size of the transmitted data. This is feasible with the *Data_Size* property, as illustrated in Listing 32 for the *Light_Intensity_SW* data type.

```
bus Data_Bus
    properties
        Transmission_Time => [Fixed => 10 ns .. 100 ns; PerByte => 10 ns
            .. 40 ns;];
end Data_Bus;
```

Listing 38: Data bus component transmission time

Note that the example we took is rather simple: AADL architectures may be much more complex, for instance by binding logical connections to virtual busses (representing communication routes) themselves bound to busses and devices representing a network.

Contribution type 3: Local communications contribution. This type of contribution is probably the less intuitive and though the main contribution to flow latency from a software architecture viewpoint. To explain how local communications contribute to flow latency, it is necessary to understand further the model of computation and communications entailed by AADL software components, in particular communications among threads and/or devices.

We decompose this explanation in two parts, answering the following questions:

1. When are data consumed/produced by threads?
2. What are the additional constraints on threads communications?

By default, the answers to these questions are:

1. Threads consume data on their input ports when they are dispatched, and produce their output at completion time (this can be refined using the *Input_Time* and *Output_Time* AADL properties). One data is consumed when a thread is dispatched (this can be refined using the *Dequeue_Protocol* AADL property), and one data is produced per activation of a thread (this can be refined using the *Output_Rate* AADL property). In the remainder of this paragraph, we keep this default configuration as this is the one we used in the AADL model of the object carrier robot, except for the *Dequeue_Protocol*: we used the *AllItem* property value meaning that the queued is emptied by the activation of a receiving thread.
2. Communications are not really constrained, meaning that threads consume the latest (set of) data arrived at dispatch time. This can be refined using the *Timing* AADL property (default value being *Sampled*). We explain how this property value impacts the flow latency analysis in the case of the object carrier robot.

Listing 36 represents the configuration of connections between process *carry_object_process* and device *light_sensor_driver* on the one hand, and between process *carry_object_process* and *left_wheel_driver* and *right_wheel_driver*.

For the latter, the value *Immediate* is assigned for the *Timing* property, which means the recipient can only start its execution when new outputs are produced by the process. As a consequence, the activation of the device always receive the latest data produced by the process.

Overall, the end-to-end latency time can be computed in OSATE: with the configuration described in this book, the result obtained is in a range of 3 to 30 milliseconds.

Other Analyses The two analyses presented in the previous sections are very common and essential for embedded systems design. However several other analyses can be conducted from an AADL model such as memory usage, power consumption [17], mass, costs, etc. On the behavior side, verification and validation can also be performed via tools such as the BLESS proof checker [179] and model checking [42, 223]. More information on analysis tools for AADL can be found at [6].

8.6.8 Code Generation

The ultimate activity to be performed from an AADL specification is automatic code generation. Once the analyses have shown a correct design, a large part of the code can be generated automatically, thus reducing the risk of errors introduced by humans. There are currently two main code generators for AADL: the Ocarina tool [222] and RAMSES (Refinement of AADL Models for the Synthesis of Embedded Systems) [235]. In this chapter, RAMSES is presented due to its refinement-based approach providing more accurate analyses and supporting automated design optimizations.

Concepts The process flow of the RAMSES approach is presented in figure 8.26. RAMSES takes as input an AADL model and transforms it into another AADL model into which details specific to an operating system specification selected by the user has been added. The POSIX, ARINC 653 and NXT OSEK are currently supported. This step is called model refinement since the generated AADL model includes details specific to the operating system and therefore is of a lower level of abstraction than the original model. The advantage of this approach as opposed to code generated directly from the initial model is a reduced semantic gap between the refined model and the code. In addition, the refined model can be further analyzed giving more accurate analysis results due to this reduced semantic gap.

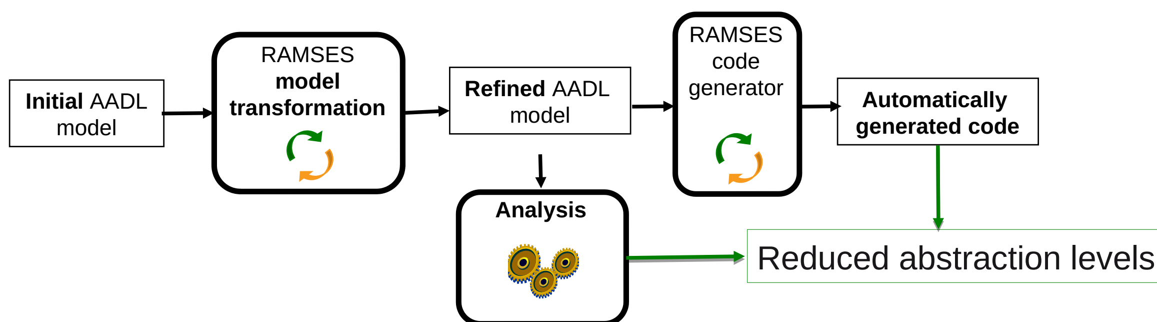


Fig. 8.26: The RAMSES process flow

The result of such analyses can be used to select of appropriate model transformations for applying design patterns to the initial AADL model for optimizing non-functional properties. The model can be refined iteratively until the desired quality attributes are achieved. Then, a code generator for the selected operating system generates platform-specific code including the required configuration files for the operating system such as thread scheduling and memory partitions.

There are many ways code can be generated from an AADL model. A first approach consists of generating skeletons from AADL components by assembling the source code set by the programming properties to subprograms and / or threads as presented in section 8.6.6. An alternative is to translate into code the behavior specification of components as specified using the AADL behavior annex. This later approach is not presented in this chapter, since the behavior annex has not been introduced. In any case, RAMSES supports these two approaches.

Running Example Specification

Refinement

RAMSES applies several refinement rules to produce the refined model for code generation. The set of applied rules varies depending on the selected operating system platform. We illustrate here one of the simplest RAMSES refinement rule for transforming event data port communications between threads into communication via shared data access. This is illustrated by the diagram of figure 8.27 where the port connection between two periodic

threads is transformed into a shared data (queue) and two shared data access connections between the threads and the shared data. For this, the event data ports of the threads have also been converted into data access features. In addition, a call to a subprogram to manage the access of the queued data has been added to each thread.

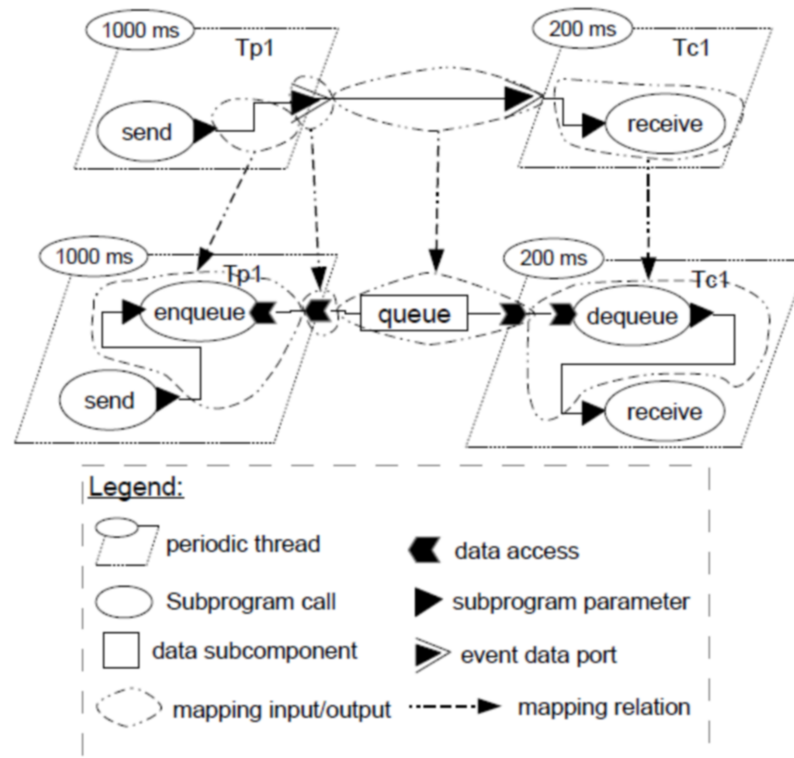


Fig. 8.27: The RAMSES local communication refinement rule (reproduced from [234])

Code Generation step

During the modelling of the software application of the line follower robot, various properties have been set to subprograms and data components for their mapping to existing C code artifacts. We briefly describe the essence of these properties using our software application example.

As shown in Listing 32, we created new data types for our software variables that map to the system variable abstract types that they extend. To these types we added the *Data_Representation* property from the data modelling annex to specify that our variables will be represented as integers in code. For the robot state internal variable, we have set the *Data_Representation* as an enumeration with values *FORWARD* and *STOP*. The *Source_Name* property specifies the name of a corresponding data structure in C code located in a C file specified by the *Source_Text* property. An initial value of *FORWARD* is set on the data subcomponent of the *Carry_Object_Process* process of Listing 36.

For subprograms such as *Compute_Turn_Angle_SW* (Listing 33), the *Source _Language* property has been set to "C" and similar to the Robot_State data component type, properties *Source_Name* and *Source_Text* are used to identify corresponding subprograms in C code.

From the AADL threads of our software application and their properties such as their priorities, as well as other properties set on the NXT ARM processor of the NXT brick (Listing 28) onto which the NXT OSEK firmware has been installed, the code generator also generates the configuration file for the NXT OSEK operating system specifying which threads should be scheduled and how they should be scheduled.

8.7 Summary

This chapter introduced the AADL language with a focus on its use for the development of CPSs. The modelling and analysis of a simple object carryer robot was used to introduce AADL by following a typical V-cycle development process (figure 8.3) augmented with some of the requirements engineering management practices promoted by the REMH (figure 8.9) and supported by the ALISA set of notations and workbench. The required AADL notions have been introduced gradually as they were required by the modelling steps of the followed development process.

This allowed presenting how AADL can be used to design a CPS with a top down approach starting from modelling the overview of the system to be built, followed by the capture of stakeholder and goals, verifiable requirements, system functions, physical plant model, software application model and deployment model. From such complete model including the software (cyber) part deployed on the hardware part, scheduling and latency analysis have been illustrated followed by model refinement, refined analyses and finally automatic C code generation with the RAMSES approach and tool.

We hope that introducing AADL this way allowed a better understanding of how AADL can be used and what it can achieve in termes of analyses, design space exploration and system synthesis via automatic code generation.

8.8 Literature and Further Reading

The AADL language and its annexes are quite rich and it was not possible to present the concepts thoroughly in this chapter due to space constraints. However, many documents are readily available for more in depth learning.

- The official AADL website [6];
- "Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language" [100] is the reference book about the core AADL language written by Peter Feiler and Dave Gluch. Peter Feiler is the main architect of the AADL;
- An introduction to the modelling of system families / product lines with AADL is given at [99];
- More information on ALISA can be found at [205, 82];
- More information on RAMSES can be found at [235].

8.9 Self Assessment

1. Write a non-functional requirement in ReqSpec for the latency in steering the robot that will constrain the results of analysis in section 8.6.7.
2. Add a predicate to verify that the latency is below the limit of 20 ms.
3. Complete the Lego Mindstorm robot product line family by adding AADL system classifiers to the robots library package of Listing 27 for the EV3 version of the Lego Mindstorm brick and robot.[2].
4. Model software requirem ents in ReqSpec derived from the example functional requirements for the system functions of section 8.6.4.
5. Perform a latency analysis for the obstacle detection function similar to the one presented in section 8.6.7 for steering the robot. First add the missing device driver to the software application system for the sonar sensor and add the required flow annotations and properties. Perform the analysis with OSATE.

[2] The EV3 Hardware Developer Kit document can be found at https://www.mikrocontroller.net/attachment/338591/hardware_developer_kit.pdf

Acknowledgements

The authors are grateful to the ISC chair (Ingénierie des Systèmes Complexes) on cyber-physical systems and distributed control systems for supporting this work. ISC is sponsored by renowned industries such as Thalès, Dassault Aviation, DCNS and DGA and operated by the École polytechnique, ENSTA ParisTech and Télécom ParisTech engineering schools.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.