

## Chapter 2

# The System Design Life Cycle

**Nikolaos Priggouris, Adeline Silva, Markus Shawky, Magnus Persson, Vincent Ibanez, Joseph Machrouh, Nicola Meledo, Philippe Baufreton, and Jason Mansell Rementeria**

This chapter focuses on a generic process for developing (safety-critical) systems. After a reminder concerning the current development process including the safety aspects, the “CESAR-proposed” development process, based on multi-views and a component-based approach, is highlighted.

Despite the fact that the system design life-cycle in each CESAR domain (aerospace, automotive, rail and automation) is characterised by many commonalities, there are also inherent differences, prescribed by domain standards, which are usually reflected in the overall engineering activities. Convergence on a generic development process is crucial for CESAR because this process is the cornerstone of the RTP. More specifically, it will condition the RTP instantiation capabilities and the data which must be manipulated at all stages of system development. These data are the main drivers for sharing information between the development phases on which the interoperability principles rely.

---

N. Priggouris (✉)  
Hellenic Aerospace Industry S.A., Tanagra, Greece  
e-mail: [priggouris.nikolaos@haicorp.com](mailto:priggouris.nikolaos@haicorp.com)

A. Silva  
Fraunhofer Institute for Experimental Software Engineering IESE, Kaiserslautern, Germany

M. Shawky  
Centre National de la Recherche Scientifique, Paris, France

V. Ibanez · J. Machrouh · N. Meledo  
Thales Group, Neuilly-sur-Seine Cedex, France

M. Persson  
KTH Royal Institute of Technology, Stockholm, Sweden

P. Baufreton  
SAGEM Defense Securite, Paris, France

J.M. Rementeria  
Fundación European Software Insitute, Donostia – San Sebastián, Spain

The cross-domain generic process depicted in this chapter is the result of close cooperation between the domain applications. This proposed process is subject to evolution throughout the CESAR innovations cycles. Indeed, only the experiments led by the application domains will be able to demonstrate its adequacy and its adaptability to meet the industrial needs.

The main breakthrough of the proposed process is that it tries to focus on, and impose the use of, models and components during all design phases so that the system description moves from a document-based approach to a more intuitive one where formal methods can be used and early validation techniques can be applied in a more rigorous way.

The chapter also includes a short presentation of important modelling aspects that are regarded as a prerequisite for addressing the “CESAR-proposed” development process.

## 2.1 The “Existing” Development Process

All industrial partners involved in CESAR deal with the development of embedded safety-critical real-time systems. This includes in the majority of cases the consideration, during the development and design life-cycle, of mechanical, electrical and software parts. At a high level of abstraction, the complete product life-cycle can be described by the activities depicted in Fig. 2.1.

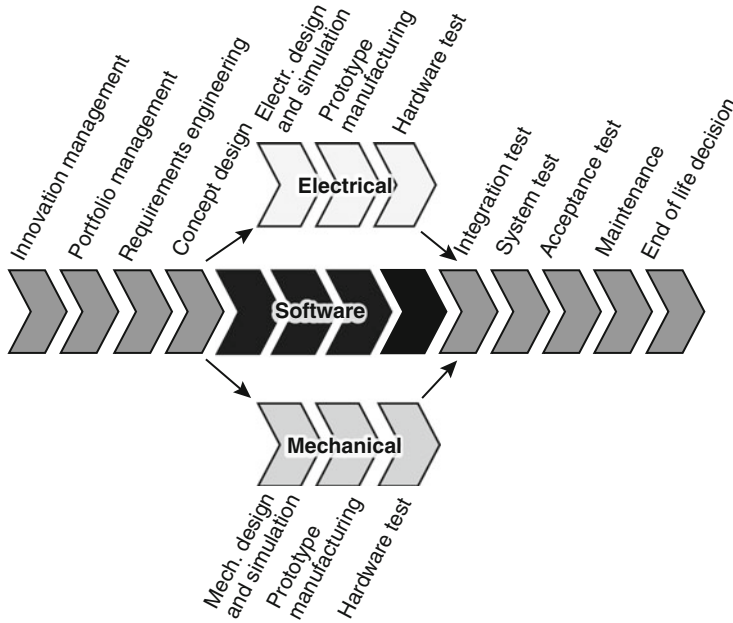
The focus of the CESAR project within this generic process is on systems development, starting with Requirements Engineering and ending with the System Test. Software related steps, which are the main focus at detailed design level, are explained in more detail below.

The V-Model depicted in Fig. 2.2 offers a finer grained view of the various steps and interactions pertaining to the development process and can be regarded as the most commonly used work-flow that applies at system or software level. It is also the basis for safety standard compliant development processes (as in for example the generic IEC 61508 [85] or in the domain specific standards IEC 50128 [49] for railway, ISO 26262 [152] for the automotive and ARP 4754 [141] for aeronautics).

In the context of software intensive system development, the mechanical, hydraulic and electrical parts of the system become environmental constraints for the electronic part of the system.

The left branch of the *V-Model* addresses mainly design and analysis phases. Although variations of the model do exist, in general the main activities performed can be summarized in the following figure (each activity is referenced by the number drawn on Fig. 2.2):

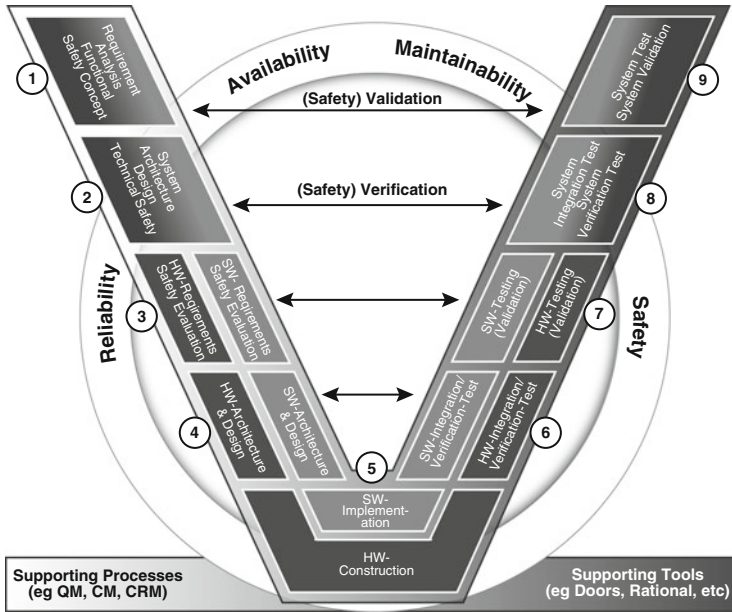
1. Requirements Analysis & Functional Safety Concept The customer requirements are analysed, elicited and refined. The missions and the functions of the system are highlighted. In parallel to this functional approach, a first level of safety analysis is conducted. This safety related activity is elaborated in the safety process related subsection under the term FHA (Functional Hazard Analysis).



**Fig. 2.1** Generic life-cycle of system development

The requirements elicitation and refinement are generally supported by tools like DOORS [81] while the functional analysis and FHA mainly rely on engineering judgement performed manually. Even if some models can be developed for functional aspects integrating sometimes dysfunctional behaviours this is the exception and not the rule.

2. **System Architecture Design & Technical Safety Concept** The system architecture design is one of the most crucial phases of the system design. It consists of several steps such as refining the functions of the system and allocating them to the different components of the system that can be either physical or software. In parallel to this design phase, the safety constraints are analysed through the PSSA (Preliminary Safety System Assessment). The system architecture design and the PSSA rely, in most cases, on manual activities and experience from previous developments. At this stage, in addition to safety constraints, the architecture solution is assessed against individual criteria such as weight, performance, interface complexity, HMI (Human Machine Interface) usability. However, these evaluations are generally supported by legacy methodologies and are not defined in a systematic manner.
3. **SW & HW Requirements Safety Evaluation** In this phase of the development, the hardware and software requirements of the system are detailed. The requirements are either textual requirements, or models. At this stage, the suppliers appear in the development process taking as inputs these software and hardware



**Fig. 2.2** The different phases of the current V life-cycle

requirements for designing the parts of the system they are responsible for. In parallel to this refinement specification phase, the safety activities are deepened through the SSA (Safety System Assessment). Even if the detailed requirements of the system in some domains are specified in formal languages, current practices show that the great majority of them are still informal and natural language based. The phase may include limited validation activities that rely, in the majority of cases, on simulation. Static analysis techniques are applicable, especially for verifying safety properties. These early validations, even if limited, allow detection of potential errors before the final product is produced thus reducing the cost and time of development.

4. **SW & HW Architecture & Design** In this phase of the system development, the hardware and software of the various equipment items constituting the components of the global system are designed. The activities that apply are similar to those conducted at the system architecture phase, but focusing on a specific component of the system: (1) refinement of the functional and non-functional hardware and software requirements including the derived requirements and (2) allocation of the software functions to the hardware components. FMEA (Failure Modes and Effects Analysis) for addressing safety concerns is also performed,
5. **SW Implementation & HW Construction** Once all the components of equipments are designed, the hardware items are physically developed and assembled and the software modules are implemented in parallel and finally integrated with

the hardware. At the end of this step the software and hardware elements should be available for verification activities. Some unit testing may be performed in parallel with implementation, especially for software. In some cases, when software is generated by a qualified automatic code generator from models, the unit tests can be suppressed (but this is still quite a limited practice).

The right branch of the *V-Model* mainly includes activities related to integration, verification and validation. Integration is performed at various levels (SW, HW, System). V & V activities rely mainly on testing, when the product or a part of the product to be verified is available. Sophisticated test benches may be used, depending on the tests to be run and on the stage of that verification is carried out at. The main activities are:

6. **SW & HW Integration/Verification Test** In this step hardware and software components are assembled. Verification tests are executed to prove compliance with the design objectives. These tasks mostly rely on reviews and analysis that are conducted by the components' suppliers.
7. **SW & HW Testing** In this step, functional tests are conducted against the hardware and software components that have already been verified against the design and passed a first level of integration. The tests aim to ensure that the software and hardware components fulfil software and hardware requirements. They usually rely on sophisticated test benches that provide a good approximation to the system functional environment.
8. **System Integration and System Verification Test** In this step, the various elements of the system (SW, HW) are combined and verification against functional and non functional requirements at system level takes place. Integration testing activities are performed, by the system integrator with the suppliers' support, on test benches that are representative of the real system environment.
9. **System Test and System Validation** This final verification phase consists of verifying that the system meets its functional and non functional requirements in its operational environment. In general, this phase is called product validation because it ensures the right product has been built.

In addition to the design and V&V activities, transverse tasks such as configuration management, traceability management and quality assurance are also carried out in order to support the exchange and coordination of work-product artefacts between the various phases.

## 2.2 The “CESAR-proposed” Component Based Development Process

As it becomes evident from the “existing” development process, a significant number of design activities are not fully supported by models. In addition, re-usability is generally not very well managed in the development process as the components,

whatever their nature, have not been designed with reuse in mind. This situation impedes the deployment of real product line capabilities and consequently acts as a deterrent in reducing development time and cost. An exception to this rule is the automotive domain. Here the state of industrial practices have demonstrated significant re-usability. Therefore, advancements in the automotive domain can bring valuable experience to other application domains.

The aim of the “CESAR-proposed” component-based development process is to provide a methodology that allows leveraging the productivity gains offered by uniting model-based development with component-based development. While the latter can contribute significantly to re-usability and minimize the need for re-verification or re-qualification activities, the former, if supported by appropriate tools, can support automation of certain tasks and identification of errors at an early design stage.

Based on the above some of the expected characteristics of this enhanced development process can be summarized as follows:

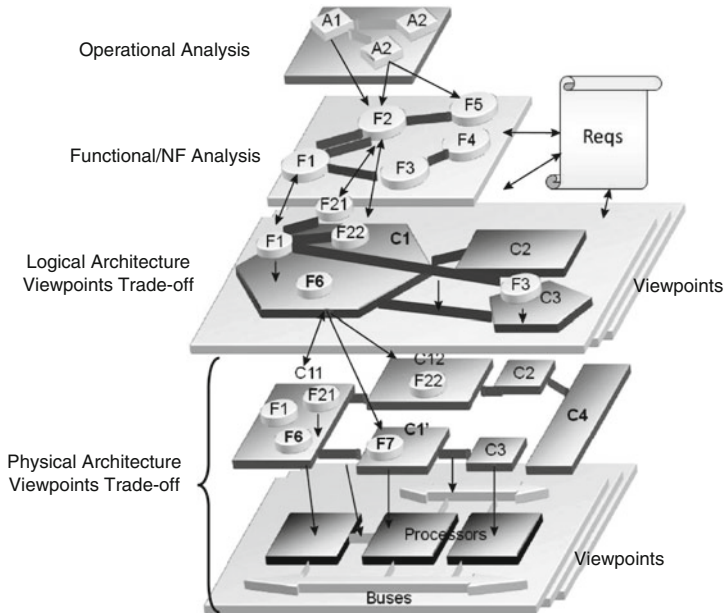
- Use of models as a basis for the development process,
- Definition of components as primary and mandatory artefacts throughout the development life-cycle,
- Traceability between development steps, requirements and various types of artefacts in general,
- Possibilities for early verification and validation (at design stage),
- An enhanced safety process based on models that are fully linked and/or synchronized with system design models.
- Adoption of product line principles during component design that can promote re-usability

Since the objective of this CESAR development process is to rely on existence of the appropriate models comprised of components, it is necessary to identify each activity of the V life-cycle process depicted on Fig. 2.2 that is to be questioned. In this section and corresponding subsections, we will propose adaptations to the design activities and the V&V activities of the previous paragraph so that they can rely on a fully component model based approach. Early validation will be promoted, as design models will be available prior to implementation of the product’s components. The new approach for safety related activities is addressed in the safety process subsection of this chapter (Sect. 2.5).

### ***2.2.1 Design Activities Relying on a Component Model***

The three primary activities that constitute the design process are:

- Functional analysis
- Requirements refinement
- Architecture design



**Fig. 2.3** The functional analysis and the architecture design phases

These activities (addressed by steps 1, 2 and 3 respectively in Fig. 2.2) are crucial for the system development and, until today, are not supported by clear and efficient engineering practices. Therefore in CESAR we paid particular attention to these steps and tried to converge on a common cross-domain vision in mutual agreement with all industrial partners.

In CESAR during system design and in order to address both functional analysis and architecture design we consider a multi-view approach where different representations of a common underlying model can be used for depicting certain features of the system under design, Fig. 2.3 gives an overview of this component-based multi-view approach. More details on how to define such a system model as well as on the proposed views and their implementation (viewpoints) can be obtained in subsequent sections (see Sect. 2.3)

In most application domains, functional analysis is formally preceded by an operational analysis where the aim is to highlight the missions of the system and to decompose these missions into operational scenarios. These operational scenarios reflect the way in which the system will be operated by the customer or generally interact with its environment. This step is considered optional. Models developed at this stage could be based on activity or use case diagrams or similar formalisms.

The functional analysis step allows definition of the functions through functional models that are built from existing elementary functions. Taking operational scenarios as inputs, it is possible to verify whether the functions defined are sufficient to accomplish the system missions. These functional models allow refinement and

formalisation of the functional and non functional requirements of the system. These requirements will preferably be formalized with the use of a RSL (Requirement Specification Language). The CESAR project defines the basic concepts and possible syntax of such an RSL (see Chap. 3). Once the functional model's associations to non functional constraints have been specified, the system architecture design can start.

Architecture design concerns the definition of the fundamental organization of a system. The word fundamental derives from the fact that this organization determines key properties such as performance and cost of the system. The system architecture design phase is formally split in two steps:

- The logical architecture definition
- The physical architecture definition

These two steps are supported by multiple views. Indeed, during system architecture design, in complex systems, many engineering teams are involved and each team is usually interested in certain system aspects. To give an example, in the avionics domain, a safety engineer is usually only interested in the functional and safety aspects of the system while a system engineer who is in charge of Integrated Modular Avionics (IMA) concepts focuses more on interfaces aspects, performance constraints, etc. Each discipline examines the system under its own prism, the so called *viewpoint*. These different viewpoints, are defined following a fully component-based approach for both logical and physical level analysis.

During the logical architecture definition, the functions are refined in sub-functions and are allocated to logical components. This process may include multiple steps of hierarchical refinement. At each step, the logical models are validated against the functional and non functional requirements, outputs of the functional analysis phase. Some different architecture solutions can be envisaged and trade-off analysis can be conducted in order to choose the best compromise. This trade-off analysis, usually referred to as *architecture exploration*, is based on multi-criteria assessment and could be considered either as an extension or as an integrated part of the overall architecture design (architecture exploration concepts are briefly presented in Sect. 2.3.4 while for more details please refer to Chap. 4).

The second step of the architecture design is the physical architecture analysis phase that consists of developing models that represent the physical parts of the system and then allocate the logical components on the physical ones. At this stage, a complete executable model of the system architecture is available. New trade-offs based on multi-criteria can be performed to compare different physical allocations and to exhibit the preferred allocation solution. Some of these trade-off criteria are proposed in Table 2.1.

Moving from system-level design to item (detailed) design we should consider:

- The hardware and
- Software architecture design and implementation.

These steps constitute significant parts of the detailed design that correspond to activities 4 and 5 in Fig. 2.2.



**Table 2.1** Example of criteria that may apply based on a particular viewpoint

Viewpoint	Homogeneous criteria definition
Safety	Minimal cut sets Fault probabilities, Absence of unexpected state
Performance	Response time, Latency, Worst Case Execution Time (WCET), Remaining bandwidth of the network, Remaining memory
Functional/logical	Number of functions, Depth of functional decomposition, Invariant functional property, Number of functions activated to accomplish missions, Number of outputs/inputs per function
Physical	System weight, Average power consumption of the physical components, Adequacy of physical link related to functional exchange between components, Number of connectors, Maximum number of parameters transmitted per physical media, Geometrical dimensions

At the end of the architecture design phase, the software and hardware requirements are completely specified. In particular, the software requirements are specified in formal models and can serve as inputs for the software parts of the equipment. In this case, and provided that development tools support it, the embedded code of the equipment(s) can be automatically generated from the software detailed models.

The software and hardware architecture design can be decomposed into logical and physical exactly as described above for the system architecture design. Indeed, equipment might be composed of different hardware components and of a set of software functions. The allocation activity from logical components to physical components follows the same principles as the ones described above. Then, the hardware and software implementation can be derived from the hardware and software models. The detailed design may also be supported by different views and corresponding viewpoints, similar to those described at system level.

The process described relies on fully formalized component models, thus avoiding ambiguities at all stages the development process. In addition, this fully model-based and component-based approach opens new perspectives in terms of V&V. This topic is discussed in the next section.

### 2.2.2 The “CESAR-proposed” Process for V&V Activities

One of the main interests of model based design is the ability to conduct validation based on models, offering the capability to detect errors early in the design process and thus avoid costly rework at later stages of the product life-cycle. A model-based approach can support an integrated V&V strategy encompassing simulation activities and static analysis on models in addition to direct testing and static verification on the final product. Figure 2.4 depicts the V life-cycle with particular focus on the V&V activities.

The main characteristics of these V&V activities are:

- All artefacts are traceable
- Lower level models are verified against upper level models
- The design is verified against requirements relying on simulation and analysis activities
- Lower level requirements are validated against upper level requirements and stakeholder requirements, to check if the requirements correctly and completely describe the needs
- Testing activities are conducted through out the development life-cycle to verify the product fulfils the design requirements and the functional product requirements.

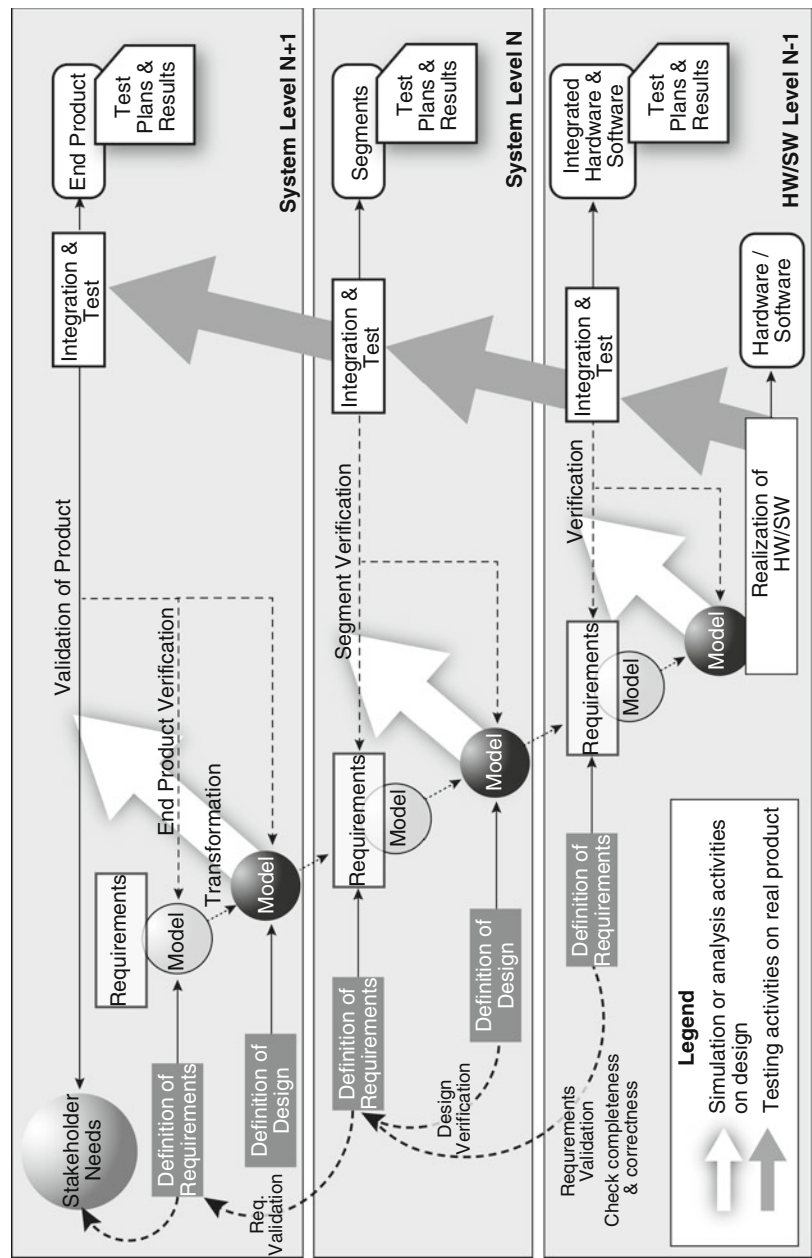
We should also distinguish between V&V activities at detailed design and V&V at system (architecture) level. At system level, analysis of system architecture correctness is performed against the designed model by checking a set of predefined criteria that depend on the particular view of interest. This *model checking* technique needs to be supported by appropriate tools. In CESAR and in order to address the V&V process the multi-view design approach is augmented with a set of criteria that are specific to one or more views and can be regarded as part of a corresponding viewpoint. System architecture assessment against a set of criteria is a major requirement to guarantee/prove that a system fulfils its non-functional constraints while meeting its functional objectives. This type of architecture assessment is called *multi-criteria assessment*.

Architecture assessment multi-criteria can be classified in two different types:

- Homogeneous multi-criteria, criteria that can be applied on the same view,
- Heterogeneous multi-criteria, criteria that have to be applied on different views.

An example of such criteria, categorized per viewpoint and identified during an initial gap analysis is given in Table 2.1.

Within CESAR, given the fact that we address safety critical embedded systems, it seems reasonable to expect multi-criteria assessment based mainly only on safety and performance criteria. Such a restriction does not mean that the other criteria will not be assessed, but it will necessitate that different multi-criteria assessments will be done one after the other. *Simulation* remains the main dynamic V&V technique but is strongly dependent on the presence of tools that can support behavioural definition of a given system and executable models.



**Fig. 2.4** A generic model-based development process with particular focus on testing

At detailed design stages, formal V&V can apply through various types of static analysis. The V&V methodologies that apply at component detailed design level are analysed in another chapter of this project handbook. Here we briefly state that static V&V analysis techniques may include model checking, theorem proving, constraint programming, and abstract interpretation, while dynamic techniques are usually test-driven and focus mainly in *automating the generation of test cases*. The testing process automation encompasses

- The capability to generate the test vectors from requirements and models automatically,
- To run the test campaign automatically,
- To deduce the verdict from the testing results and the oracle (the software that evaluates the result from the test as either success or failure) automatically

## 2.3 CESAR Modelling Aspects

In complex systems, such as those designed in the domains of aerospace, rail, automotive and automation, different specialists from various disciplines interact to build a common product.

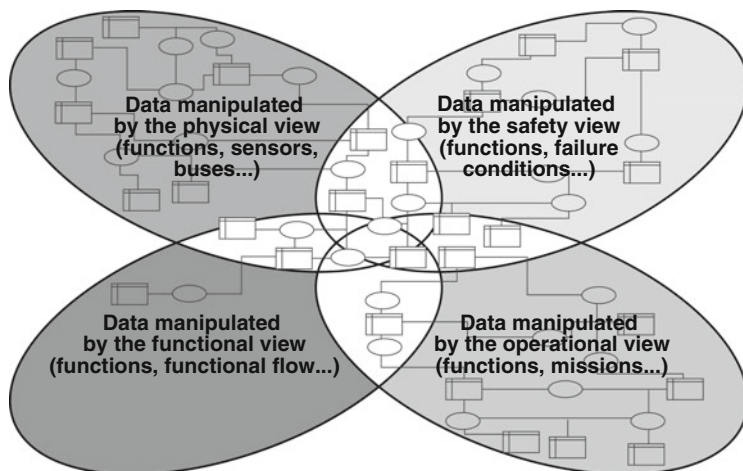
A lot of data is manipulated by several designers for different purposes. The different designers manage their activities in parallel implying, most of the time, a duplication of the same data. Sometimes, the same information is referred to using different names depending on the designer managing the information. In such a context, ensuring data consistency all along the development life cycle is extremely difficult and may lead to discrepancies that are costly to resolve.

The essential purpose of a system engineering data model is to ease collaborative work on data which is shared between several designers and crucial for their disciplines, by enabling unambiguous sharing of data and ensuring that data is properly updated when necessary.

In a model-based approach, workload is significantly reduced if it is possible to manipulate the design information through the modelling languages and exchange various models between different tools.

Therefore, the main requirements regarding model-based design in CESAR can be summarized in the following principles:

- Capability to build models relying on multiple formalisms
- Capability to couple and replace existing models in a transparent way
- Capability to design models for reuse
- Ensure consistency of data managed by the used modelling languages and formalisms
- Ease data exchange between the tools relying on usage of data shared among each other (i.e. the same global model or common set of artefacts)



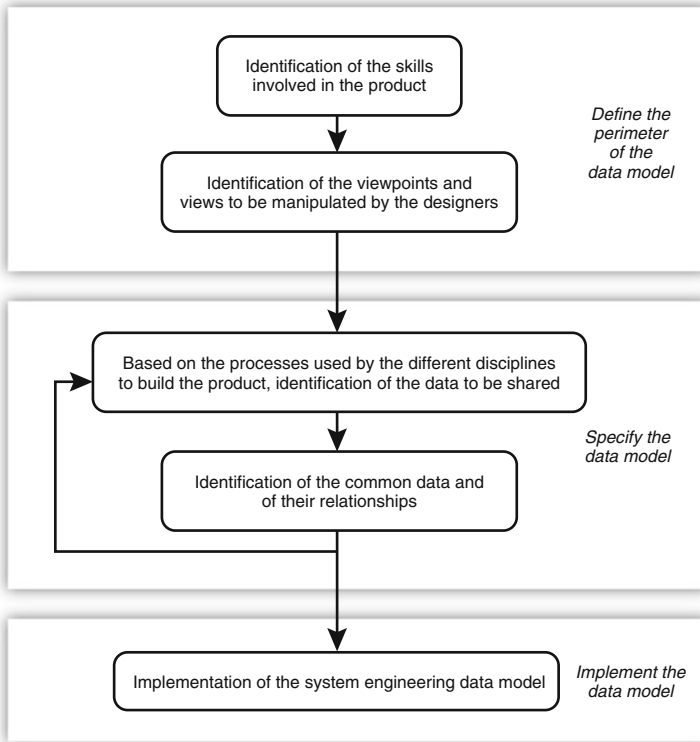
**Fig. 2.5** An example of a system engineering data model

All of the above imply that a generic system engineering data model should exist. This data model should be capable of managing the system design information, support multiple views for architecture modelling but also provide the ability to assess various architecture solutions based on multiple criteria. This model in its core will include all the common part of development data managed by the different disciplines throughout the development life cycle activities but should also be extensible in order to easily adapt to future needs. The core part should reflect the most critical data aspects, that is the ones that if modified may have a strong impact on the whole product. In Fig. 2.5 an example of a system engineering data model for aeronautics is shown. The data in white sectors correspond to data shared between at least two views. The core data model part corresponds to the elements residing in the common intersection between all views.

Such a system engineering data model though, capable of manipulating development data managed by different engineering teams and process activities, is also dependent on the application domains and on the industrial processes used. Generally, it seems very difficult to build up a system engineering data model that covers all the domains involved in CESAR: Aerospace, Automotive, Rail and Automation. Even inside each domain, the system engineering data model may vary depending on company specific industrial processes. Therefore, a generic approach must be set up to specify a system engineering data model. *This approach should be customizable by each company according to domain, internal procedures and specific project needs.*

The different steps needed to define a system engineering data model are shown on Fig. 2.6.

The first step consists of identifying the different skills that have to work in tight cooperation. From this identification of disciplines, the views and the associated



**Fig. 2.6** Workflow for defining a system engineering data model

kinds of models to manage in a model based engineering approach are deduced. This phase allows pointing out all the actors that will have to work together.

The second step is crucial as it deals with the identification of the potential data to be shared. This potentially shared data is identified by analysing the processes used by each discipline to develop the product. In particular, the outputs of activities that are produced by one discipline and consumed by a set of activities led by other disciplines are potential candidates for sharing.

After having identified the potential data to be shared, the core data used to build the foundations of the system design activities are selected as common shared data and will be part of the system engineering data model. In order to succeed in this approach, all the stakeholders (or at least the majority of them) have to be involved.

Once the data to be shared have been specified, the system engineering data can be implemented. The implementation details depend on the language used to describe the data model but may also be affected by the environment in which this data model will be used.

As it becomes evident that the data model is based on a viewpoint driven approach, different views are defined that can be used to represent a system from the perspective of a related set of concerns.

The following subsections will try to further expand on many of the concepts pertaining to the CESAR engineering data model.

### ***2.3.1 An Engineering Data Model to Support Architecture Modelling***

In CESAR a high level system engineering data model has been proposed. This conceptual data model has the ambition to cover the activities led in the system architecture design as already presented in Fig. 2.3. Therefore, this data model describes the different concepts and details the expected relations between these concepts, to support the design activities along the steps of a system development process. It addresses the following four types/levels of analysis:

- Operational analysis
- Functional and non functional need analysis
- Logical Architecture analysis
- Physical Architecture analysis

The expected model elements for each type of analysis together with their relationships are summarized in the next subsections. A “UML class diagram like” formalism is used. Any implementation meta-model (as supported by RTP tools) shall allow the user to develop, transform, analyse, exploit a model defined with these concepts and their relationships. Any used model transformation tool shall be able to extract these concepts and links from models built using the RTP. Support for most of these concepts is provided in the CESAR Common Meta Model (CMM). The CESAR Common Meta-Model (CMM) (see Sect. 6.4) includes all these system modelling concepts as part of its core meta-model. CMM also provides additional meta-model concepts to address other engineering aspects such as management of requirements (see Sect. 3.5).

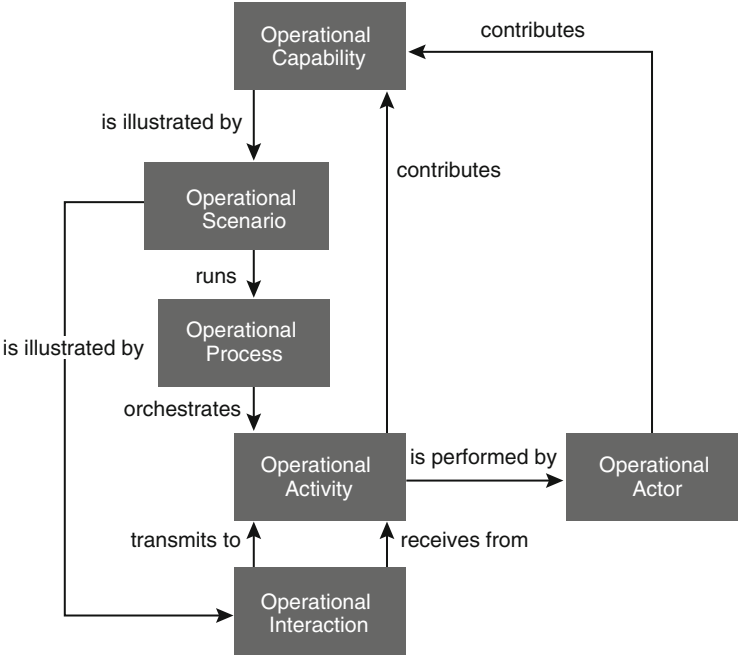
#### **2.3.1.1 Operational Analysis**

This step focuses on analysing the customers’ needs, the goals that shall be reached by the system users, and the activities being performed by the users.

Outputs of this step leads to an “operational architecture” describing and structuring the needs, in terms of actors/users and their operational capabilities and activities, operational use scenarios giving dimensioning parameters, operational constraints including safety, security, system life cycle, and others.

The elements that are introduced in this step are:

**Operational actor** Person or entity, including system end users, involved in activities and missions related to the use of the system



**Fig. 2.7** Relation between elements in operational analysis level

- Operational capability** Quantified overall operational goals, objectives, services that are expected from actors
- Operational activity** Any process step or function, both mental and physical, performed by actors toward achieving some objective
- Operational interaction** Generic exchange or information flow between operational activities
- Operational scenario** description of the sequence of interactions between actors in a given context, usually to achieve a given capability
- Operational process** Sequence of activities contributing to an operational capability

The relation between all of these elements is described in Fig. 2.7.

**2.3.1.2 Functional and Non-functional Need Analysis**

The role of this step is to focus on the system itself in order to define how it can satisfy operational needs (prescribed in operational analysis) along with its expected behaviour and qualities. This includes the definition of functional as well as non-functional requirements of the system, e.g., safety, security, and performance. At this step, they are focused on system boundary level and not on



individual system components. Furthermore, role sharing and potential interactions between system and operators are of concern here.

Outputs of this step mainly consist of the system functional need description, descriptions of interoperability and interaction with the users and probable external systems (functions, non-functional constraints, and interoperation), and system/SW requirements. Note that these two steps, which are a prerequisite for architecture definition, have high impact on the future design being developed in the forthcoming steps, and therefore should be approved with and validated by the customer.

The elements that are introduced in this step are:

**System** An organized set of elements functioning as a unit, which is the target of the engineering.

**System actor** External actor interacting with the system via its interfaces

**System capability** Contribution of the system to an operational capability

**System function** Actions or services defined on a system or an actor

**System port** “entry point” or means to interact with a function or an actor

**Data** Data related to functional and non-functional need, especially functional exchange contents

**Functional exchange** Data exchange between functions, linking their ports

**System functional chain** Sequence of Functions, activated through a path and carrying non functional properties such as latency, criticality level

**System scenario** Description of the sequence of interactions between system and actors in a given context

**State or mode** The state characterizes the system behaviour in given environmental conditions. A mode or state is activated by external conditions and by operators

The relation between all these elements is described in Fig. 2.8.

The relation between the operational analysis and the functional analysis is described in Fig. 2.9. As a general rule, each element in the functional analysis has to support the equivalent element in the operational analysis.

### 2.3.1.3 Logical Architecture Analysis

The role of this step is to identify the system’s parts (hereafter called components), their roles, relationships and properties, while excluding implementation or technical issues. This constitutes the logical architecture of the system.

This is achieved by grouping or segregating functions according to functional and non-functional constraints, and afterwards allocating them to logical components. During this step major functional and non-functional constraints (safety, security, performance, etc.) are taken into account. Usually, not all of the non-functional requirements may be supported in an equally good manner, so different variants have to be analysed and the best compromise has to be taken.

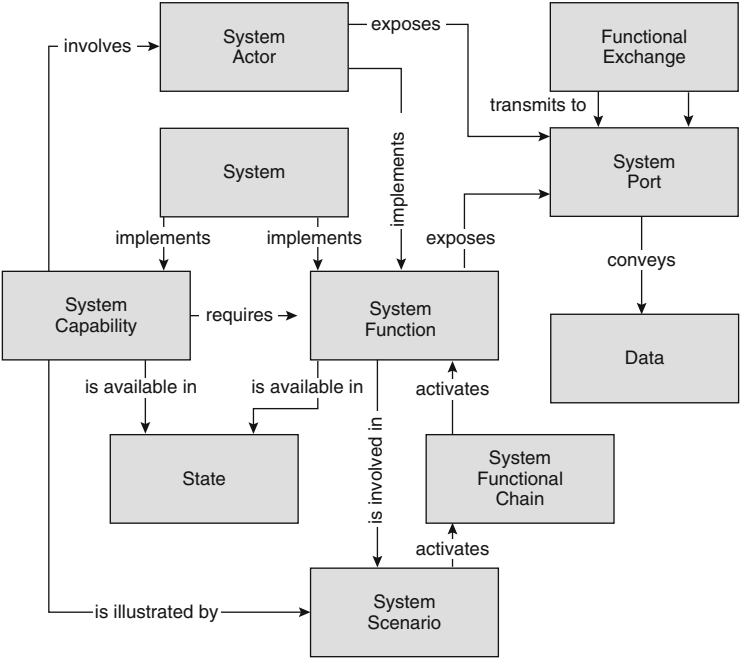


Fig. 2.8 Relation between elements in the functional analysis phase

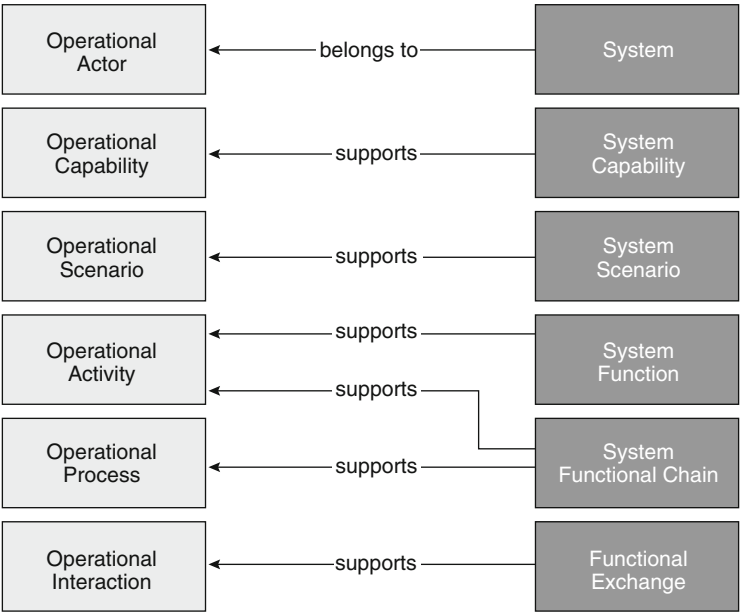
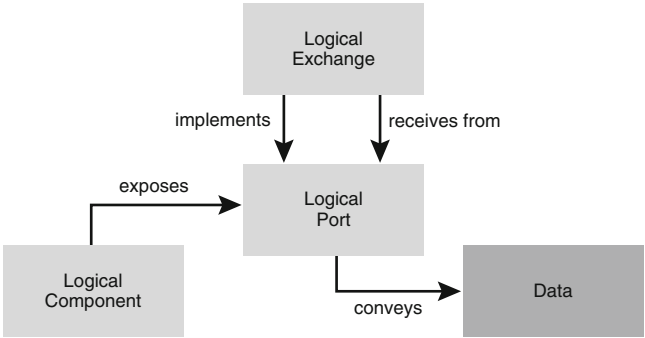


Fig. 2.9 Relation between operational analysis and functional analysis



**Fig. 2.10** Relation between elements in the logical analysis phase

The output of this step is a logical architecture consisting of components, their interface definitions, and functions allocated to them (along with their functional exchanges).

Since the architecture has to be validated against the needs identified during operational and functional analysis, traceability links to the requirements and operational scenarios are established. The elements that are introduced in this step are:

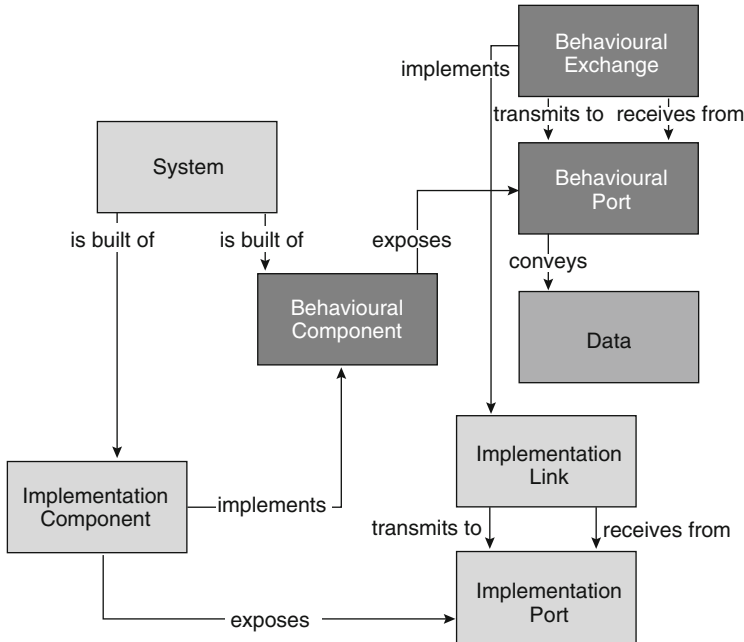
- Logical component    component present in an abstract, notional breakdown of the system. This kind of component is independent of technology choices (i.e selection of hardware architecture).
- Logical functions    functions that are allocated on logical components
- Logical exchange    exchanges between logical components
- Logical scenario    scenario between actors and logical components
- Logical functional chain    functional chain allocated on logical components

The relation between all these elements is described in Fig. 2.10.

**2.3.1.4 Physical Architecture Analysis**

This step defines the “final” architecture of the system at physical level, ready to be developed (by lower engineering levels). Therefore, it introduces rationalization, architectural patterns, new technical services and components, and makes the logical architecture evolve according to implementation, technical and technological constraints or choices (at this level of engineering). Again, non functional aspects are taken into account as in logical architecture, and multiple different variants are being analysed and checked against each other.

Output of this step is the chosen physical architecture consisting of components to be produced and the way they are taken into account in the subsystem design. As is the case with logical analysis, traceability links towards the requirements and operational scenarios are established.



**Fig. 2.11** Relation between elements in the architectural analysis phase

The elements that are introduced in this step are:

**Behavioural components** part of the system that realizes a set of functions.

Examples include software component, FPGA VHDL program etc.

**Behavioural exchange** exchange between behavioural components, linking some of their ports

**Implementation components** resource hosting behavioural components.

Example: Processing unit, display unit

**Implementation link** Link between implementation components

**Physical scenario** description of the sequence of interactions between actors and physical components in a given context

**Physical functional chain** Functional chain allocated on physical components

The relation between all these elements is described in Fig. 2.11.

### 2.3.2 Views and Viewpoints

The envisioned system modelling approach is viewpoint-driven. According to IEEE 1471 [86], the definition of views and viewpoints are as follows:

**Views** A representation of a whole system from the perspective of a related set of concerns.

**Viewpoints** A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

Thus, a view normally conforms to a viewpoint and may consist of one or more architectural models. Moreover, an architectural model may participate in several views. These different models describe the view in different perspectives. In CESAR the four main views of interest are defined according to the four types of analysis described in Sect. 2.3.1.

The IEEE1471 standard however says little about how to choose viewpoints from which to construct the system views. The reason is simple: it is hard to decide on a standardized set of views that will cover all stakeholders' concerns, and still not leave many systems horribly over-specified (and hence also making architecture development more expensive due to work overhead). Many different viewpoint frameworks have been proposed as such standard sets, e.g., the 4 + 1 view model [101], MODAF [162] or DODAF [168], but none of them have been established as a de-facto standard across several domains.

Further, many issues such as safety concerns, may be approached in different manners depending on the application domain. Hence, there are many possible selections of sets of viewpoints, and the choice of views ultimately depends on the type of system under development and the stakeholders that the system developers consider most relevant. The viewpoints that will be used to describe a system are typically chosen early during the design, and maybe according to some standard, be company-specific or even produced specifically for the exact system under design. For the types of system in relevance to CESAR, clear candidates for choice of viewpoints, besides the ones defined to directly support the architecture analysis steps (operational, functional, logical, physical), are for example safety, cost, performance, product line, and so on.

It becomes evident that in order to identify a first set of viewpoints it is important to collect and categorise the issues addressed along a typical engineering cycle. Table 2.2 provides a summary of such issues based on brainstorming performed among CESAR industrial partners. Each column represents an example of viewpoint (such as safety or performance) and each row describes the engineering phase and the content that must be included in this viewpoint at each engineering phase. The proposed set may not be exhaustive and will be insufficient for particular designs or domains, however the purpose is merely to illustrate the concepts of a possible framework that can be extended and customized rather than be exhaustive.

### 2.3.2.1 Viewpoints Standardized in the CMM

Before concluding this section, a brief description will be given on how system modelling is achieved in CESAR through the use of views and viewpoints. The CMM defines *abstraction levels*, *perspectives* and *aspects*. Abstraction levels cover the certain refinements of the system beginning with the first very abstract represen-

**Table 2.2** Main concepts handled by example viewpoints along the engineering cycle

Viewpoints				
Engineering phases	System description			
	Decomposition	Behavior	Safety	Performance
Operational Analysis	<ul style="list-style-type: none"><li>• Actors</li><li>• Activities</li><li>• Exchanges</li></ul>	<ul style="list-style-type: none"><li>• Scenarios between actors</li><li>• Phases, missions</li><li>• Modes and states</li></ul>	<ul style="list-style-type: none"><li>• Feared events, failure conditions</li></ul>	<ul style="list-style-type: none"><li>• Operational latency constraints on activities</li></ul>
Functional and Non-functional Need Analysis	<ul style="list-style-type: none"><li>• Expected functions</li><li>• Data</li><li>• Data exchanges between functions</li></ul>	<ul style="list-style-type: none"><li>• Scenarios between system and actors</li><li>• Functional chains</li><li>• Modes and states</li></ul>	<ul style="list-style-type: none"><li>• Criticality of functional chains, data, and data exchanges</li><li>• Failure and recovery scenarios</li></ul>	<ul style="list-style-type: none"><li>• Latency on functional chains</li><li>• Processing complexity of functions (or full behavior description)</li></ul>
Logical Architecture Analysis	<p>Same plus:</p> <ul style="list-style-type: none"><li>• Added functions</li><li>• Components implementing functions</li></ul>	<ul style="list-style-type: none"><li>• Scenarios between system components and actors</li><li>• Functional chains allocated to components</li><li>• Modes and states of components</li></ul>	<ul style="list-style-type: none"><li>• Redundancy paths</li><li>• Vote, monitoring components behavior</li><li>• Dysfunctional behavioral model for functions and components</li></ul>	<ul style="list-style-type: none"><li>• Estimation of function resource consumption</li><li>• Estimation of data and exchanges complexity, rate</li></ul>

Physical Architecture Analysis	<p>Same plus:</p> <ul style="list-style-type: none"><li>• Implemented functions (incl. technical)</li><li>• Behavioral components (e.g. software)</li><li>• Implementing functions</li><li>• Implementation components (e.g. computers) delivering resources to behavioral components</li><li>• Interfaces between behavioral components</li><li>• Physical links between implementation components</li></ul>	<p>Scenarios between system components and actors, using interfaces between components</p> <ul style="list-style-type: none"><li>• Functional chains allocated to components</li><li>• Modes and states of components</li></ul>	<p>Same plus:</p> <ul style="list-style-type: none"><li>• Component failure scenarios</li></ul>	<ul style="list-style-type: none"><li>• Estimation of function resource consumption</li><li>• Estimation of data and exchanges complexity, rate</li><li>• Resource available for each implementation component</li><li>• Bandwidth available for each physical link</li><li>• Resulting resource consumption of functions, behavioral components . . .</li></ul>
--------------------------------	---	---	---	--

tation and ending with the final concrete representation. Perspectives and aspects define different views for the system under design. Each abstraction level generally contains a (non-empty) set of perspectives. Perspectives represent the system in certain development phases. The CMM differentiates five different perspectives that map to the type of analysis defined in Sect. 2.3.1:

1. *Operational perspective* addresses the operational analysis view
2. *Functional perspective* addresses the functional analysis view
3. *Logical perspective* addresses the logical analysis view
4. *Technical perspective* addresses a subset of elements covered by the physical analysis view
5. *Geometric perspective* (experimental) maps to another subset of elements covered by the physical view

For concerns which are cross-cutting to abstraction levels and perspectives the term aspect is used. Aspects contain representations which might belong to different abstraction levels and/or perspectives. Through aspects dynamic behaviour of a component can be classified and described.

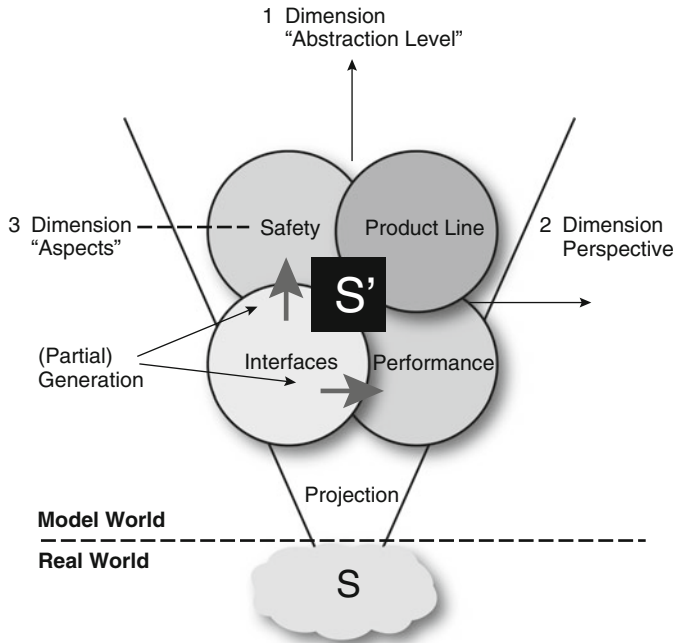
Therefore, in a model-focussed world we can regard the various models that describe a real world system to evolve along three axes covering the analysis levels of the process steps and the viewpoints. This is illustrated in Fig. 2.12). The first axis corresponds to an ordered set of abstraction levels. These abstraction levels generally start with the definition of very coarse models and end with models that address low level details of the system. Models defined at a lower level of abstraction refine models of a higher abstraction level. Several kinds of composed models can be regarded on one abstraction level, which cover different perspectives of the system as a second axis of the model evolution. The models of the different perspectives may be related to each other via allocation links (check Fig. 2.28). Furthermore within the models several non-functional aspects are regarded which are covered by different artefacts of the models. Such non-functional aspects may include safety, performance, costs or interfaces depending on the kind of model and the level of abstraction. The coverage of aspects in the models therefore provides a third axis of model evolution.

Figure 2.13 provides a conceptual example from the avionics domain that illustrates how abstraction levels, perspectives and aspects are related to the development items. Only perspectives which are relevant in a certain abstraction level are contained in the specific abstraction level. The same for aspects, only the relevant items from several perspectives are associated to the specific aspect.

### 2.3.3 Components and Design-by-Contract

CESAR design methodologies and suggested modelling approach also strongly promotes component-based engineering (CBE) principles during design and construction of safety-critical systems. This will lead to reusable elements that adhere to

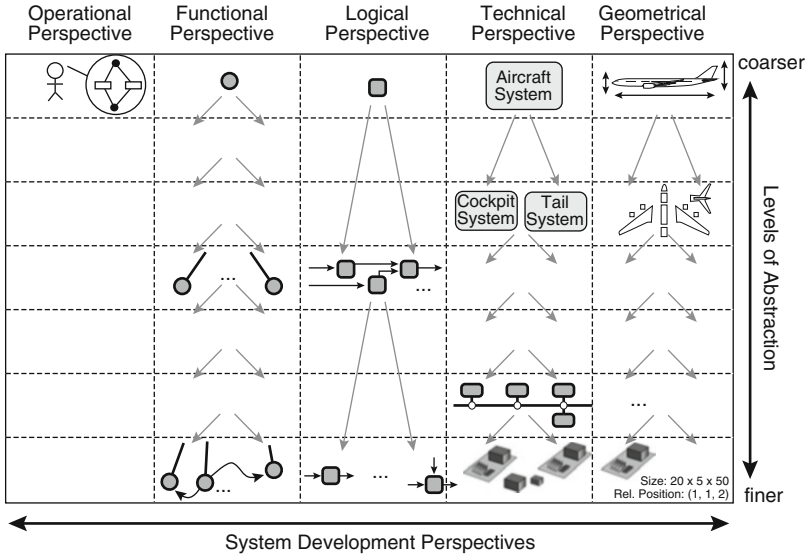




**Fig. 2.12** System model evolution along three axis (perspectives, abstraction levels, aspects)

clearly defined specifications and will eventually provide for significant benefits in terms of cost, time and reliability. The components can be maintained in a repository where they are constantly improved and validated in order to attain a level of pre-qualification.

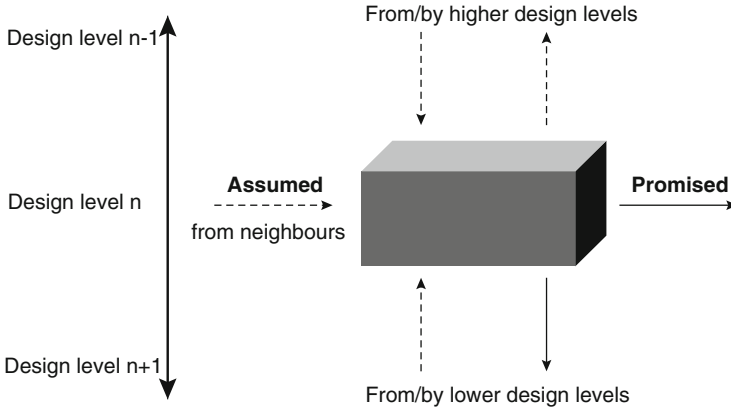
According to [128] the CBE paradigm should encourage the use of predictable architectural patterns and standard infrastructures that improve overall product quality. It encompasses two parallel engineering activities, domain engineering and systems' component-based development (CBD). Domain engineering explores the application domain with the specific intent of finding functional, behavioural, and data components that are candidates for re-use and place them in re-use libraries. CBD elicits requirements from the customer and selects an appropriate architectural style to meet the objectives of the system to be built. The next steps are to select potential components for re-use, qualify the components to be sure they fit the system architecture properly, adapt the components if they must be modified to integrate them, and then integrate the components into subsystems within the application. Therefore the expected process contains two parallel life cycles, one that relates to the building of a component and its deposition in a repository and another one that address system development through assembling and reuse. Chapter 5 related to component-based development provides additional information on CBD concepts and proposes an adaptation of the classical V cycle (called W model) to efficiently address the two life-cycles mentioned above.



**Fig. 2.13** Example of abstraction levels and perspectives structuring (Graphic taken from [SPES] [39])

Component-based approaches “gain from” the definition of unambiguous component interfaces, and the separation of component interfaces, and their realization. Furthermore, components need to be self-contained, realizing exactly their offered services. These realizations need to strictly conform to the defined component interfaces – based on this principle, component realizations may be replaced without affecting other components, as long as the new realization does not affect a component’s interface. Interfaces therefore have a very important role in component-based approaches. They do not only cover operation names, parameters, return values, and exported variables, but also need to provide a description of the offered services. For safety-critical systems the sole existence of an interface is not enough. There are convincing arguments for adding further semantic annotations to component interfaces, to make their usage safer. Essentially, an interface signature alone does not tell us what the interface methods are supposed to do and that this information is necessary to use the interface safely. It becomes evident that in a component-oriented development approach, there is a need for a way of determining beforehand whether we can use a given component within a certain context. Ideally, this information would take the form of a specification that tells us what the component does without entering into the details of how. Furthermore, the specification should provide parameters against which the component can be verified and validated, thus providing a kind of *contract* between the component and its environment (which can include interactions with other components).

The contract theory and the notion of contract were originally introduced for object oriented designs (mainly class methods but in general they can be applied



**Fig. 2.14** Assumptions and promises for a component's contract

to functions) in the design-by-contract principle [116, 117]. One could summarize design by contract by “three questions” that the designer must repeatedly ask:

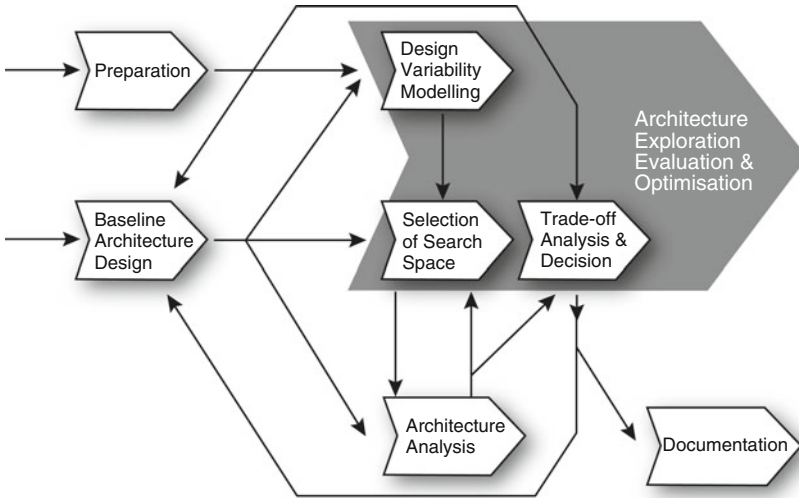
- What does it expect?
- What does it guarantee?
- What does it maintain?

Contracts therefore do not only define detailed component specifications, but also define necessary assumptions (pre-conditions) that need to hold in order to ensure proper behaviour for a given component (post-conditions). These assumptions and promises are usually specified on a per component basis. An assumption specifies how the context of the component (i.e. the environment from the point of view of the component) should behave. Only if the assumption is fulfilled, the component will behave as promised. Figure 2.14 shows the relations between assumptions and promises.

In the CESAR project, the component models proposed (i.e. HRC model or X-MAN model) support the notion of contracts through the definition of pre- and post-conditions that can also be used for component formal verification and validation. More details are provided in Chap. 5.

### 2.3.4 Architecture Exploration

A useful feature when designing system architecture is the ability to explore different architecture alternatives in order to choose the one that corresponds to a reasonable compromise between different system properties. Considering system development with given end-user requirements and use cases, a designer may have many options at hand, including the choice of suitable functional artifacts and behaviors, choice of hardware elements and structure, and many solutions for mapping behavior to the structure.



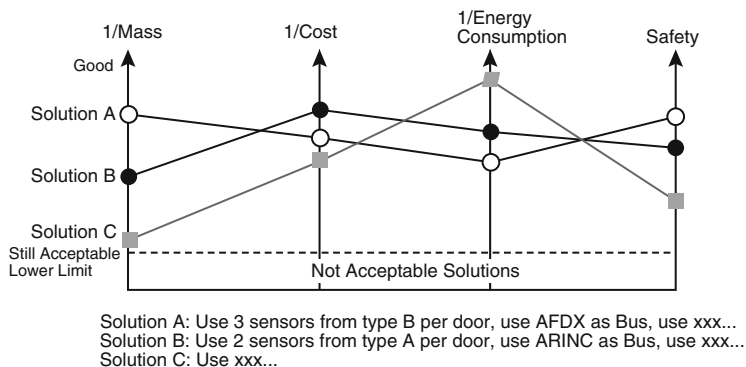
**Fig. 2.15** A simplified view of the architecture exploration methodology developed in CESAR (the full version is described in Sect. 4.3).

In general, each requirement can be met through several different alternative solutions in system design and implementation. In practice however, the choices can be more restricted due to earlier design decisions. For example, a fixed network infrastructure will set limitations on the feasible allocation of software components onto hardware nodes due to bandwidth and delay constraints. Further, some functions may already be associated with suppliers that deliver integrated components (hardware and software), implying no freedom of allocation.

Whenever alternative behaviours, structures or mapping solutions are possible, it is also the task of architecture designers to elicit and describe the design options, and to provide the selection of which option(s) is(are) preferred. While many design alternatives can often be identified and resolved based on stakeholder consensus, heuristics, or design patterns, explicit architecture exploration and evaluation allows more systematic investigation of the merits of different alternative architecture solutions.

*Architecture exploration* is a design approach aiming to support the deduction of architectural design alternatives for full or partial system optimization. It is covered in further detail in Chap. 4. It allows a designer to systematically examine viable and alternative design options for a particular system, and thereby complements the application of heuristics and domain experiences for justifying design choices and performing design optimization. Figure 2.15 offers an overview of the related work products and activities, which are further elaborated in the next paragraphs.

In architecture exploration the system architect is responsible for selecting choices from a number of alternatives based upon criteria based on metrics such as performance, cost, memory resource usage, mass, failure rates and so on. When the



**Fig. 2.16** An example trade-off analysis illustrating how different solutions affect metrics and hence indirectly criteria

selection has taken place, the architect should be able to understand the implications of certain design choices and do some trade-off analysis. Figure 2.16 attempts to visualize how different solutions (on the vertical axis) can impact the metrics associated with certain trade-off criteria (on the horizontal axis). The final choice will depend on the aggregation of these metrics and the possible weights assigned to them.

To put architecture exploration into its context in the CESAR project, architecture exploration can easily be coupled with a multi-view approach to modelling, since the architecture exploration typically has to consider several different properties of the proposed architecture that are typically not (all) strongly coupled with each other. Example properties that need to be addressed in safety critical systems may relate to reliability/safety, performance, complexity/extensibility, cost, weight, power consumption, or any other well-known or custom metric that can be defined for a system. All of them should be considered in the context of the various CESAR viewpoints as already presented in the previous subsection.

The design space available to a developer depends on the particular scenario, and technical and strategic constraints. Several options may be evaluated in parallel or sequence, especially if it is unclear which the most suitable product design might be. For example, consider the common case of adding a new function to an existing product range. The question then arises whether new computer units need to be added, or whether the new function can be accommodated by existing computer units. The overall design space in this case consists of alternative hardware structures (including options with new hardware elements and their physical placement and cabling) and alternative allocations of the functions to existing and/or new hardware. On a more detailed level, the new functions need to be structured, partitioned and scheduled on existing and new resources; there alternative realizations might also exist.

By describing design variability and thereby capturing the design variants and their impacts on system attributes, the design space for the exploration work will

be described. In any real-world setting, a selection and prioritization between these design variants will need to be done, since it is not feasible to evaluate them all. A smaller search space (a subset of the design space) will need to be selected, either at the start of the process, or iteratively using heuristics. Formalization of the design-space allows the automation of exhaustive exploratory search of some or all possible alternative architectures resulting from different design variants in scope. This kind of exploratory search of alternative architectures can for example be supported by genetic algorithms.

To support architecture exploration, efficient tools for architecture analysis, producing values for selected metrics for evaluation, will be needed. These tools will assess the viability of each architectural design from the chosen viewpoints and provide metrics for comparing them in regards to the chosen functional and quality goals. Analysis can be performed at different levels of abstraction or different development life-cycle stages, for system level features or operations, functional and logical design, software and hardware solutions. The analysis tools may require dedicated analysis models (e.g. component behaviours, end-to-end timing, system safety and reliability). For example, while a timing analysis requires models that capture the triggering, periodicities, execution time of behaviours in functional chains, a safety analysis requires models that capture the subsystems' failure modes, errors and error propagation. Each architecture analysis normally assesses only one architecture instance at a time.

Trade-off is a natural part of any multi-criteria architecture exploration, just as any design task with conflicting goals.

Whereas in current practice, many architectural trade-off decisions are taken on the basis of best practices and application domain experience, resolving design variations in safety- or mission-critical systems often would be improved by substantiated decisions with explicit evidence supporting the rationale. In order to compare and choose among substitutable architectures, architecture design needs to define the optimization goal(s), evaluation criteria and metrics.

For alternative architectures, trade-off analysis aids designers in establishing a choice. There are both quantitative (mathematical optimization) and qualitative approaches to trade-off analysis.

## 2.4 CESAR Reference System Engineering Process

Considering the cross-domains dimension of CESAR, it is necessary to offer flexibility in the definition of the development process. One of the foundations of the RTP is the ability of its adaptation to industrial needs while promoting sound engineering practices. Thus, within CESAR a development process definition shall be able to be built either from scratch using process elements or relied on pre-existing process model possibly enhanced and tailored depending on the industrial context.

The method chosen to compose a reference system engineering process is by assembling “method fragments” or “chunks” defined and stored in an appropriate method library repository. The language chosen to support this compositional process approach was SPEM 2.0 [122]. The Eclipse Process Framework (EPF) [47] Practice Library (EPL) [54], a UMF-compliant [165] method library that is available from the Eclipse project, was used as the base for the expression and collection of these “methods chunks”. These process fragments collection, along with other material, constituted the CESAR Practices Libraries (CPL).

In order to represent the CESAR practices in a modular, extensible and reusable way, the concept of “Library” has been investigated. A “Library” of practices represents an organised and easily accessible collection of the techniques, methodologies and processes that are being developed in the project. The Unified Method Architecture (UMA) [164], used in EPF, is a process engineering meta-model that defines schema and terminology for representing methods consisting of method content and processes. UMA provides a clear separation of Method Content definitions from its application in Processes. This is accomplished by separately defining:

- Reusable core *Method Content*, in the form of general content descriptions such as *Roles*, *Task*, *Work Products* and *Guidance*;
- Project-type specific applications of Method Content in the form of *Process* descriptions that reference Method Content.

Method Content provides step-by-step explanations of how specific development goals are achieved independent of the placement of these steps within a development life-cycle. Processes take these Method Content elements and organize them into a sequence that can be customized to specific types of projects. For example, a software development project that develops an application from scratch performs development steps similar to a project that extends an existing software system. However, the two projects will perform similar steps at different points in time with a different emphasis and perhaps individual variations.

UMA allows each Process to reference common Method Content from a common Method Content pool. Because of these references, changes in the Method Contents will automatically be reflected in all Processes using it. However, UMA still allows overwriting certain method-related content within a Process as well as defining individual process-specific relationships for each Process Element (such as adding an additional input Work Product to a Task, renaming a Role, or removing Steps that should not be performed from a Task). Further technical details and descriptions, useful to deeply understand the CPL structure and to gain knowledge on how to author it, can be obtained from [47, 164, 165].

Using the tools and frameworks described above, several “method chunks” were authored in EPF and were used in order to define an example development process instance. The purpose was not to define a de-facto reference process applicable to all CESAR domains but rather to propose an example process covering the whole system engineering life-cycle and comprising of activities and tasks. This process would act as a starting point for application domains to create their own custom

Presentation Name	Index	Predecessors	Type	Planned	Repeatable
CESAR Reference Process	0		Delivery Process	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Design the System	1		Iteration	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Define Stakeholder Needs	2		Activity	<input type="checkbox"/>	<input type="checkbox"/>
Capture Stakeholder Needs	3		Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Identify stakeholders	4		Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Elicit stakeholder requirements	5		Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Operational Analysis	6	3	Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Requirements Definition	7	6	Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Define the constraints on system solution	8		Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Define activity sequences	9		Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Identify the interactions between users and the system	10		Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Specify critical quality Requirements	11		Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Requirements Analysis	12	7	Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Requirements Agreement	19	12	Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Perform Functional Analysis	20	2	Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Functional Hazard Assessment	21	20	Activity	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Identify Functions	22		Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Identify and Describe Failure Conditions	23	22	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Determine Effects	24	23	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Classify Effects	25	24	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Assign Safety Requirements	26	25	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Identify Supporting Material	27	26	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Identify Compliance Verification Method	28	27	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Design Architecture	29	21	Activity	<input type="checkbox"/>	<input type="checkbox"/>
System Decomposition	30		Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Allocation of Requirements	31	30	Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Evaluation of Architecture	32	31	Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Define Assessment Criteria	33		Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Assign Weights to Criteria	34	33	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Establish performance parameters	35	34	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Evaluate performance parameters	36	35	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Determine best solution	37	36	Task Descriptor	<input type="checkbox"/>	<input type="checkbox"/>
Definition of Interfaces	38	32	Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Define Component Specification	39	29	Activity	<input type="checkbox"/>	<input type="checkbox"/>
Implement the Components	40	1	Iteration	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Develop Complex Electronic Component	41		Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Develop Software	42		Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Develop Simple Component	43		Activity	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Integrate the System	44	40	Iteration	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Verify the System	45	44	Iteration	<input type="checkbox"/>	<input type="checkbox"/>

Fig. 2.17 Work breakdown structure of Reference Development Process defined in EPF composer

instances by extending or modifying existing practices or even defining their own based on specialized domain needs. In the Fig. 2.17 the work breakdown structure for the CESAR System Engineering reference process. In this example a practice is highlighted and it is shown how this practice is used (or better re-used) in a wider process. For more details on the elements comprising the Reference Development Process please refer to [31].

## 2.5 Safety and Diagnosability Process

Safety activities are an essential part of safety critical system's design which seeks to identify certain system constraints that must be addressed in order to avoid situations that may cause faults in system operation and consequently endanger life of human beings.



A first step in the safety process is to identify hazards and risks associated with new concepts or product changes. If the risks are unacceptably high, then the first steps would be to (1) try to eliminate the hazard altogether; (2) to reduce the risk; and finally (3) to provide alternative measures that minimize the impact and consequences of an accident. Given steps (1) and (2) it is clear that safety considerations strongly influence the system development including the architecture design, see e.g. Storey [149].

System requirements, including high level safety requirements, constitute inputs to the architecture design process. Initial hazard and risk analysis will yield a categorization of functions and systems into so called safety integrity levels (SILs) that characterize the severity (worst case consequences) of potential failures. The various domains' safety standards have somewhat different approaches on how to do this. The SILs imposes a number of requirements and recommendations for the consequent development that relate to:

- *Quantitative* (probabilistic) targets corresponding to the desired risk reduction (e.g. in terms of  $10^{-9}$  failures/h for highly critical failure modes) and for error detection coverage (to prevent latent faults). This kind of quantification and reliability prediction is accepted for random faults having only hardware as source.
- *Qualitative* techniques that have to be applied and judgement that have to be made in order to demonstrate and ensure a proper development and verification process with prescribed design styles such as the use of redundancy and diversity. By these means systematic safety (hardware design and manufacturing/software/installation errors) is addressed.

The high-level safety requirements (including the SIL level) will be used to define more detailed architectural requirements, and it is then the objective of architecture design to develop solutions that meet those requirements. As part of the architecture design, the safety standards will also require that independence between functions/systems of different criticality can be argued and ensured.

On the other hand, *diagnosability* is also an essential property that determines how accurate any diagnostic reasoning can be on a system given any sequence of observations. The diagnosability process can be seen as an extension of the notion of *observability* when considering the presence of potential faults but also as a generalization of *testability* since it makes use of tests that provide data for all measurable variables, e.g., for all sensors of a given system. However, limiting diagnosability to a particular observability property – defined as the capability to distinguish any two different cases of faults in a specified fault model (or at least, any two ones for which the subsequent recovery or repair processing is different) is likely to be insufficient to be usable in practice because the needed observable information depends on the diagnosis procedure. Within CESAR diagnosability issues have been slightly tackled. In the corresponding diagnosability process subsection we present some principles of a proposed analysis that can apply.

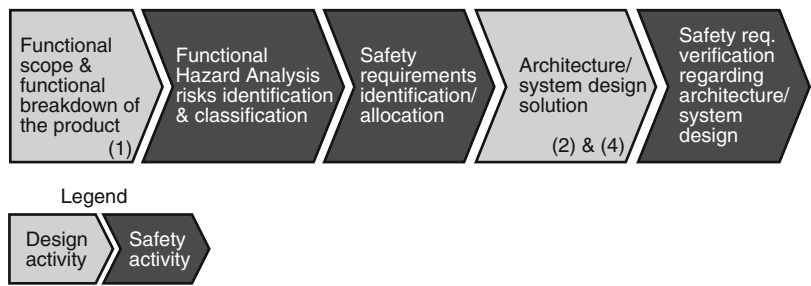


Fig. 2.18 The safety process and design process

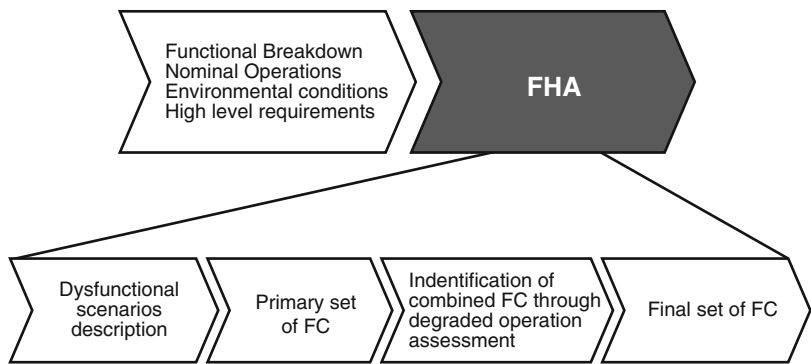


Fig. 2.19 FHA steps

2.5.1 An Overview of the Safety Process

Safety tasks are an area where significant divergence exist between the various CESAR domains especially regarding the rigour of the applied process (e.g. in avionics and rail safety regulations and standards impose many design restrictions while in the automotive and automation the safety process is considerably less restrictive).

Figure 2.18 describes the main steps of a generalized safety process interlaced in the design process (the numbers drawn on Fig. 2.18 refers to the V life cycle process phases depicted in Fig. 2.2). Each safety task is described in more details in the next paragraphs.

2.5.1.1 Functional Hazard Analysis

The main steps of Functional Hazard Analysis (FHA) are described in Fig. 2.19.

The first step consists in identifying for each function, the scenarios that lead to a failure mode considering each product mission and under each relevant environmental condition. For each functional failure mode, the repercussion on the

function of the failure mode is assessed and classified according to a number of criteria including:

- Total Loss of the function,
- Partial Loss of the function,
- Inadvertent functioning of the function,
- Erratic functioning of the function.

Then, an exhaustive description of the functional scenarios that induce abnormal behaviours is elaborated. The description of the abnormal behaviour encompasses the description of the functional failure, the effects on the system, the procedure to deploy in order to minimize the failure consequences and the safety classification that depends on the safety standard. Finally scenarios are classified according to their repercussions and factorized per effects in order to get the failures conditions (FC).

### **2.5.1.2 Safety Requirements Allocation**

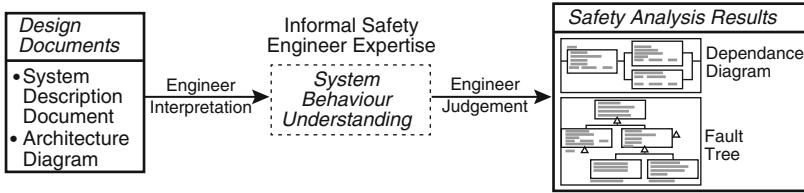
During this phase, the failure conditions identified in the previous phase are allocated to systems (that constitute the global product) which support the function. These failure conditions(FCs) define the safety requirements for each system. The system shall be designed in a way to comply with the qualitative (e.g. no single failures) and quantitative (probability values) safety requirements defined by the FC classification.

### **2.5.1.3 Safety Requirements Verification (PSSA/SSA)**

Once the system is designed, a demonstration that the system fulfils the qualitative and quantitative safety requirements is required. This demonstration should highlight that failure combinations for system components, leading to an allocated failure condition, are acceptable. Fault tree or reliability block diagrams are usually used to get cut sets and verify qualitative criteria and quantitative objectives.

During the system design life-cycle, there are two main assessment activities that try to assess system safety based on various defined qualitative or quantitative criteria. These are Preliminary System Safety Assessment (PSSA) that usually applies directly after the initial architecture has been defined and System Safety Assessment (SSA) that is the final activity that takes place after a system is integrated by its sub-elements to ensure that all safety constraints have been correctly addressed.

Note that for systems with high reliability requirements it is generally impossible to demonstrate by testing that the requirements have been achieved. Acceptance and certification are based on reliability models for the hardware and use of appropriate development techniques for the software. Process measures and special design techniques are employed to provide confidence in the software.



**Fig. 2.20** The current practice for safety engineering process

### 2.5.2 “Existing” Safety Process

Today the majority of safety activities, risks identification and safety requirements verification are performed in a completely manual manner with minimal support of tool automation. The safety engineer has to handle non formalized data and also write dysfunctional scenarios or fault trees relying mainly on engineering judgement as depicted in Fig. 2.20:

This manual safety engineering process has important repercussions on the system development in terms of cost and time. Indeed, due to increase in system complexity, the number of failures combinations becomes very huge and difficult to manage without adequate tools. As a consequence, the PSSA and SSA activities are very complex to lead and are more and more time-consuming and costly. Due to these complexity factors, it is risky to modify an existing design when developing new system features and conservative assumptions are often preferred to innovative solutions. After design changes, the update of the safety assessment induces a huge work that has a negative impact on development time and cost.

### 2.5.3 “CESAR-proposed” Safety Process

Due to the strictness and constraint imposed by the various safety standards it is difficult to develop a completely new safety methodology within CESAR. Therefore focus was mainly given in:

- Enhance traceability and consistency of safety requirements to design elements
- Enhance system modelling with safety related information and dysfunctional models (models that describe erroneous behaviour)
- Provide means to automate the generation of safety analysis data

#### 2.5.3.1 Enhance Traceability and Consistency of Safety Requirements to Design Elements

The RTP provides appropriate infrastructure services such as the *link repository* (see Sect. 6.3), implementing a lightweight integration mechanism that enables

full traceability between requirements, design elements, test procedures and results. Although this feature is not limited to safety requirements only, the safety process greatly benefits. In addition to that a couple of CESAR tools and tool-chains have demonstrated features like architecture and functional consistency checking of formalized safety requirements against a given architecture design.

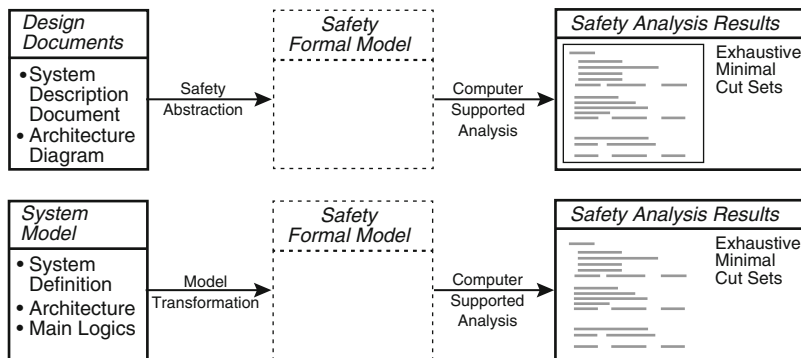
### 2.5.3.2 Enhance System Modelling with Safety Related Information and Dysfunctional Models

In order to support the safety process objectives as they have already been analysed, the CESAR modelling approach has prescribed a safety view (see Sect. 2.3.2). The safety view is supported as an extension package to the core CMM specification and it should be regarded as an *aspect* since its concerns span different perspectives. The elements of the safety view extension provide means for the analysis of potential component failures including among other concepts that allow specifying *failure conditions*, *safety cases*, *cut sets*, etc. The CMM Validation and Verification package provides also elements like *VerificationCase*, *VerificationProcedure*, *VVStatus* and others that are an important part of a safety viewpoint (safety standards always prescribe verification and validation as parts of a safety analysis). The *Verification-Case* allows specifying which component is verified and how this is done via the *VerificationProcedure*. Figure 2.21 provides an overview off all the safety concepts specified in the CMM.

The safety view is applicable at different stage of the development process. In particular in the concepts and requirements phase, in the design phase and in the validation and verification phase. Three different categories of models are involved which address three main categories of objectives of safety analysis:

- *Structural safety models*: Structural models, used to represent the organisation of the various elements composing a system, can be extended to represent how the faults and failures affect the various elements of the structure, and how these faults and failures propagate along the structure. Such augmented models are good candidates to support, possibly through coupling with classical existing safety and dependability analysis tools, the automatic analysis of fault propagation and demonstration of the related requirements. Of course, not all the characteristics of fault propagation can be easily represented on structural engineering models (e.g., thermal, EMC, etc.). Therefore these models are not expected to address all possible safety issues.
- *Behavioural safety models*: behavioural models are often used for formal verification of behavioural properties i.e., the correctness of the behaviour even in case of complex behaviour with many interactions between a large number of elements and a large number of possible combinations of events and states. It is worth investigating the applicability, efficiency and maturity of behavioural modelling approaches as a support to demonstrate e.g., fault





**Fig. 2.22** Example concept of automating generation of safety analysis data

tolerance, correctness, liveness and safety properties; despite the fact that fault tolerance is generally regarded as a particularly difficult case in this respect, and moreover the severity of consequences of potential failures of the fault tolerance mechanisms may be very high.

- *Logical safety models*: within the scope of the CESAR RTP, logical safety models can be used to describe the logical structure of the arguments (safety objectives, claims, assumptions etc.) involved in a safety analysis.

A complete safety view should encompass structural, behavioural and logical models and also provides the means to capture the safety abstraction concerning system dysfunctional behaviour that is the basis for analysing failure and faults propagation.

### 2.5.3.3 Provide Means to Automate the Generation of Safety Analysis Data

Formalising system requirements and extending existing system models with safety concepts and dysfunctional behaviour, are the main prerequisites in order to enable automation of certain safety tasks including among others:

- Partially or completely generation of failure propagation models from system description
- Creation of initial PHA and FHA table
- Minimal cut sets/fault tree generation

These could be achieved by first translating system models (augmented with safety data) to formal safety models and then applying computer aided analysis tools to generate the safety analysis results (see Fig. 2.22). Some of the expected benefits of such an automatic translation are:

- Reduction of iteration time improving the interaction between safety and design activities,
- Earlier safety assessments improving architecture design trades-offs,
- Possibility to assess a greater number of alternative architecture solutions,
- Capability to generate automatically minimal cut sets,
- Reduction of errors risk due to the automation of fault tree generation and the initialization of the dysfunctional models,
- Ability to focus on system dysfunctional behaviour that can be shared easily with other domains
- Facility to update the cut sets after modifications to the model design.

Materialization of such automations are still quite limited in the CESAR project. The RTP platform includes tools chains that can be used for model-based safety analysis. This includes, e.g., a model checker which identifies the minimal cut sets from Simulink models exported in CMM format. Creation of initial PHA and FHA tables is possible from simple Rhapsody Models as well as partly generation of fault trees. However the overall provided functionality is still quite immature and should be regarded more as a proof of concept rather than being applicable in real system development.

### ***2.5.4 An Overview of the Diagnosability Process***

Usually in safety critical systems we refer to *fault diagnosability*, which includes:

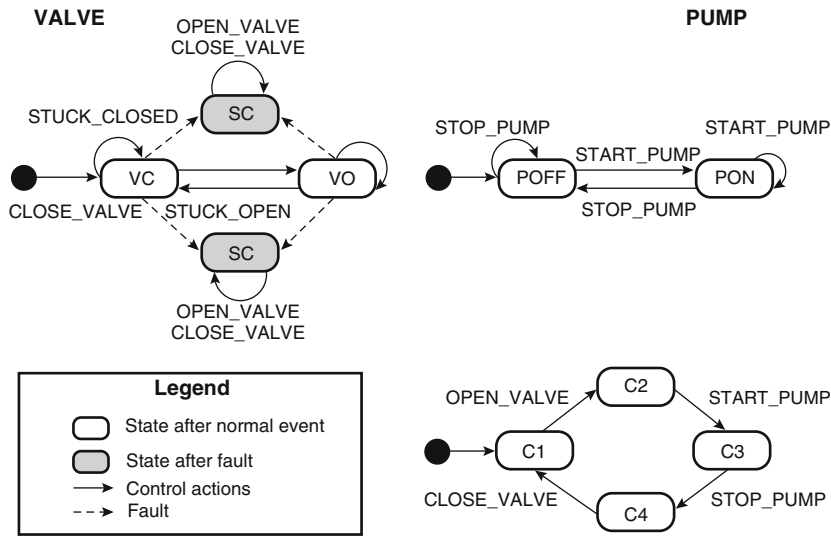
- The ability of a system to support the identification of information related to its potential faults/errors or
- Capability of a system and its monitors to exhibit different observables for different anticipated faulty situations

The system designer must ensure that the system is diagnosable, i.e. must ensure that the errors that will appear are identifiable by appropriate procedures. A diagnosability procedure can be considered as a form of test that checks/measures the values of multiple variables and applies algorithms to determine if the system (whether it is SW, HW, etc.) functions correctly. Diagnosability should be addressed during design in order to provide a more reliable system, with predictable maintenance costs.

In order to perform a functional diagnosability analysis, a behavioural model of the system is needed. Multiple models can be envisioned such as continuous-state-based, event-based, or hybrid.

When dealing with the diagnosability of discrete event systems, the most widely accepted analysis method is the “diagnoser” approach Sampath [134]. The input model is a finite deterministic state machine modelling the system behaviours (a representative example is given in Fig. 2.23). The analysis is performed by augmenting that model with non-normal or fault states and checking if the augmented and transformed model is diagnosable.





**Fig. 2.23** Example behavioural components models of a PVC (Pump, Valve, Controller) sub-system of HVAC (Heating, Ventilating, Air Conditioning) system

While the system model being functionally diagnosable and the existence of a diagnoser are necessary conditions for the final embedded system to be diagnosable by itself, they are not sufficient. The interaction between the functions and the diagnoser as well as their mapping to the underlying architecture has to be considered. It is thus necessary to define checkable properties that the architecture has to demonstrate, with respect to the functions and the diagnoser. To give an example, in order to check properties of the functional architectural diagnosability, the system architecture should be described in an architecture description language such as AADL (Architecture Analysis and Description Language) or a hardware description language such as VHDL (Very high speed integrated circuit Hardware Description Language) or SystemC. The main idea is to enable parsing of the architecture description model to extract and identify all architectural features necessary for the analysis, such as embedded computing units, sensor devices, communication links, data flows as well as inputs/outputs connections representing state and event variables.

In fact, each state and event variable found in the functional models (automata) can be matched to the corresponding data flow from the AADL model, including origin, destinations, physical links supporting the data or message transmission, etc. The architecture properties are evaluated using that matching. For instance, to check the accessibility property for a given event variable, one has to check that a data flow exists for that variable, (i.e. it is actually output by some component), that the data flow actually reaches the observer (i.e. there exists a connection between its source and the observer) or that its source and the observer are mapped to the

same computing unit. The global accessibility property is evaluated by repeating that process for each event variable, one at a time. Checking for other properties follows a similar process.

This approach can possibly be automated, making its integration into a tool-chain possible. The CESAR projects provides only suggestions on how to address diagnosability issues. In this sense tools, such as the COSITA prototype Khelif [99], (a tool that enables analysis of iterated co-simulations traces in order to evaluate the interaction of functions with the architecture according to selected critical scenarios), although still a work in progress, are good candidates if diagnosability is to be considered as a metric during the architecture exploration process.

## 2.6 Product Line Engineering

Product line engineering (hereafter referred to as PLE) aims at building products for a well-defined problem space in shorter time, with less costs and with higher quality through a proper utilization of platform assets, including product line architectures (blue-prints for creating families of related applications), reusable components and design artefacts as well as process related artefacts and certification arguments.

In CESAR PLE activities, the book *Software Product Line Engineering* by Pohl et al. [126] is used as a foundation, as it represents a well-established reference in the area of software PLE and is also aligned with the EU-supported PLE projects ESAPS [53], CAFE [65] and FAMILIES [60] of the ITEA programme [87]. It should be noted that, despite that the focus of [126] is on software, the presented approaches have been applied to hardware products as well.

The rationale behind applying PLE is to save cost and improve efficiency by two main effects:

- *Improving reuse*: The reuse of development assets was also the goal of previous reuse approaches. Compared to previous approaches, however, PLE goes further: it involves the *business case of the products* and plans explicitly for reuse. Previous approaches have covered reuse on a technical level only; organizational hurdles of the company or particular demands and restrictions resulting from the business case have not been taken into account. PLE now means engineering for reuse that covers not only technical aspects but also integrates organizational, management and business case views.
- *Managing variability*: A product line provides the basis for products that differ for each customer and/or application. These products are built from a given asset base – that is, what forms the product line. However, since each customer will require specific modifications making the product a (more or less) unique solution, the producing enterprise must efficiently build the product according to the given requirements. The more efficient a product can be derived from the product line, the more competitive can the product be sold. The main advantage

for applying PLE is the structured and systematic handling of variability that improves the efficiency of the derivation (configuration) process.

To cover these two issues, PLE aims at the systematic development of a set of similar systems by understanding and controlling their common and distinguishing characteristics.

PLE is based on a development process including both, development *for* reuse (domain engineering) as well as development *with* reuse(application engineering). These two processes are detailed in the next sections.

### 2.6.1 Product Line Engineering Life-Cycle

Pohl et al. [126] have established a representative framework addressing the PLE life-cycle which was used as a reference for analysing the current practice within CESAR. According to this framework, the product line engineering activities are divided into:

**Domain engineering** is the part of the product line engineering process in which the commonality and the variability of the product line are defined and realized. **Application engineering** is the part of the product line engineering process in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability.

Figure 2.24 shows the general set-up of this framework which is explained in details in the next paragraphs.

The PLE life-cycle starts with a product management activity that defines the concrete boundaries of the product line in terms of the envisaged product or the product categories that are to be supported. This activity is in close relation with *domain engineering* and often considered as part of it.

#### 2.6.1.1 Domain Engineering

*Domain engineering* is the part of the product line engineering process, in which the basis for the reuse is developed. This means, that the commonality and the variability of the products in the product line are identified or defined and that the common parts of the product line products are realized. Domain engineering comprises therefore the following activities:

- Product line scoping
- The development of the product line variability model
- The development of the building blocks for the product line's products – the core assets
- The definition of the production plan

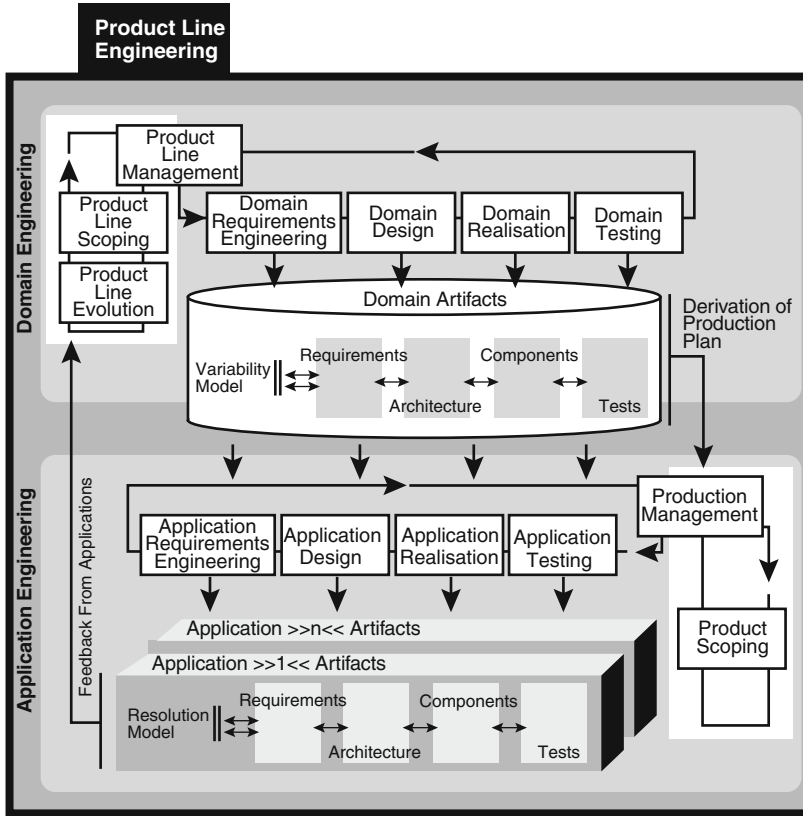


Fig. 2.24 Software Product Line Framework (Pohl et al. [126])

The scoping of the product line will define the products planned to be built by the product line and decide on the approach to be followed. There are two primary strategies, proactive and reactive. In the case of a pure proactive approach, all of the products needed on the foreseeable horizon are supported by the product line. In the case of a pure reactive approach, only the products needed in the immediate term are supported by the product line and new products are incrementally added as the needs change. There is, of course, a continuum between the two. The scope is not static; it is continuously changing in order to allow the product line evolution.

The variability model is created based on the scope of the product line and continuously evolves throughout the assets' base creation, which is supported by usual development steps, including requirements engineering, design, realization and testing, with the entire product line in mind. Concurrently with the development of the domain assets the production plan is derived that contains a description on how the different assets can be used to yield a product.

All the entities developed in the domain engineering process are called domain artefacts and form the product line's base of reusable assets (also known as *core asset base*).

### 2.6.1.2 Application Engineering

*Application Engineering* is the part of the product line engineering process, where the products are actually built, by reusing the set of assets defined during the domain engineering process. The way the applications are built will depend on the development process. Two extra activities, however, need to be performed:

- Variability model resolution, according to the product scope
- Core assets adaptation, according to the production plan

As Fig. 2.24 shows, assets developed in particular domain engineering tasks influences the corresponding application engineering tasks and its resulting artefacts. Reusable artefacts produced during domain engineering (that is, the domain artefacts), are made available for the different application engineering instances that yield the different products as members of the product line.

Resulting from different areas that variability covers, resolving the variability is considered possible at different stages during a system's life-cycle:

- *At Modelling Time*: Variability resolution is done when generating a model that either includes a determined feature or not. The generated model can serve as a building plan for later creation stages as well as for simulations.
- *At Assembly Time*: Variability is resolved at the time when either (a) the source code is written or generated, or (b) the physical construction of a device takes place.
- *At Deployment Time*: At deployment time, variability can be handled by deployment settings that configure the system itself. Or, binding at deployment time refers to settings that configure the deployment environment.
- *At Run Time*: At run time, a system can adapt dynamically to an environmental change or a new setting it has detected. For example, for software systems a plug-in mechanism is possible that allows binding functionality during run time.

There is also a *feedback path* from application to domain engineering where specific application engineering findings are propagated to the domain models and used to update them. This may happen when problems emerged during the reuse of certain domain artefacts, that is, if defects were observed or change requests occurred. Another possible reason may be that it becomes necessary to incorporate newly identified generic aspects of the domain models or new variability characteristics. Thus, the domain and application engineering tasks are *iterative* – meaning that they can be performed more than once in order to achieve the desired results. This completes the development cycle of product line engineering and makes it evolutionary.

## 2.6.2 Variability

*Variability* plays a major role in product lines while many of its aspects apply to other engineering activities addressed by CESAR, such as architecture exploration. Therefore, it is important to explain some major aspects of variability that will help to better understand CESAR introduced improvements through the RTP platform PLE related tool-chains.

### 2.6.2.1 Variability Characteristics

Individual products of a product line share common parts. These common parts are the commonalities. But naturally, the products show also differences, that is, they show variability, which is expressed by so-called variation points. Actually, variation points identify the points where products of a product line family show differences. From a system development point of view, *variation points* thus identify situations where decisions about the configuration of the individual products have to be made. In this context, a *variant* denotes then a particular setting for a variation point – after a certain decision was made. Variation points can occur in many ways in products since variability may span system characteristics related to structural architecture, behaviour, functional and non-functional aspects.

Pohl et al. characterise variation points by three main dimensions [126]:

- *What does vary?* This question should lead to the identification of the item or property of a system that represents a variation point.
- *Why does it vary?* This question should deliver the benefit or reason of the variation. Examples of reasons are regulatory demands in different market regions.
- *How does it vary?* The answer to this question leads to the variability subjects and denotes the actual occurrence of the variability.

Variability can also be characterised as negative or positive [35]. *Positive variabilities* leave the underlying commonality untouched – they always add something to the commonalities. *Negative variabilities* break commonalities – they remove properties that are usually common. Negative variabilities embody the “same as-except” concept [14], which means that there exists a quasi-common part which is overridden by a negative variability in exceptional cases. This results in an optimization, because less variation points are needed. The choice between negative or positive variability should be driven by the effort that is required to build the product line members.

Pohl et al. also distinguish between *internal* and *external* variability, denoting that internal variability is not visible to the customer while external is.

For embedded safety relevant systems, hardware and software variability should not be considered separately for software and hardware. Thus, it may prove

worthwhile to reduce hardware variability and provide the needed variation in functionality through software variants (or vice versa).

### 2.6.2.2 Variability Management

One of the main activities in PLE is variability management. Variability management can in general be divided into two areas: *variability realization* and *variability specification*.

Variability realization deals with the implementation of the different types of variation points in the development artefacts (e.g., specifications, architecture, code, V&V artefacts). For example, variability realization may entail the implementation of generic test cases that contain variation points in terms of parameters and decision nodes. It may also entail the creation of generic specifications in terms of documents that contain place-holders for the points that may vary.

On the other hand, variability specification concentrates on describing, structuring and managing the variation points that are spread out across the different development artefacts. Variability specification can be seen as a configuration layer over all variation points in the development process. In other words, the variability specification manages all information that allows the artefacts' variabilities to be controlled. This configuration layer can be seen as a front end for navigation through the different variations in the development process and for resolution of the variation points. Hence the variability specification shows only configuration information, that includes:

- (a) The type of variability implemented by a variation point,
- (b) The position of the variability in the global hierarchy of variations and, most important,
- (c) The dependencies between variations

In order variability specification to be beneficial, it is necessary to establish clear traceability links between the configuration possibilities shown in the variability specification and the different variation points arising in the development process.

There are several definitions and approaches to deal with Variability Management, most of them applied or envisioned to be applied in Software Product Line (SPL) engineering processes:

- *Feature modelling (FM)*: this approach is used in order to model both the commonalities and variabilities of product variants. The main idea behind this formulation is the representation of the variation through the classification of features within different categories (e.g. mandatory, optional, alternative) using a graph-based model [18]. This approach is useful in high abstraction engineering levels, since it provides a global perspective, serving as a basis for both, end-customer and engineering configuration. However, the main drawback of this approach is the increasingly complexity that it is inherent to features definition.

As soon as the feature model covers several aspects of the domain, the model itself becomes unmanageable.

- *Decision modelling (DM)*: Decision models focus only in the variable part of the product variants. This means that they guide the derivation of new products, and control their development through the definition of inter-dependencies and relationships among the decisions. Indeed, decision modelling provides great flexibility and precision regarding the capture of decisions and their relationships at different levels of abstraction in a more precise way compared to feature modelling. In this sense, complex dependencies can be easily addressed and at the same time, the complexity grows less than in the previous approach since only the variable is managed. The main idea behind decision modelling is to define the domain based on decisions (business or technical ones).
- *Common Variability Language (CVL)* [140] is a generic and separate language for modelling variability in models in any Domain Specific Language (DSL) that is defined based on Meta Object Facility (MOF) [115]. In the CVL approach three kind of models are used:
  - The base model (a model described in a DSL).
  - The variability model (the model that defines variability on the base model).
  - The resolution model (the model that defines how to resolve the variability model to create a new model in the base DSL).

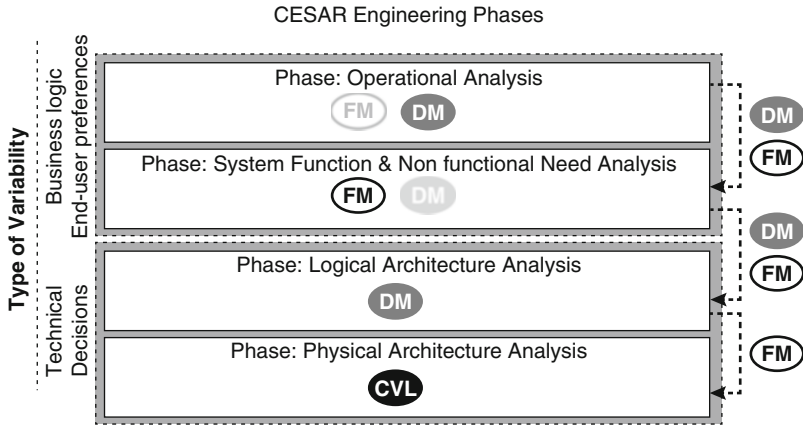
It must be noted that there are some overlaps among the identified approaches, which means that in certain cases is feasible to use multiple of them, although not recommended. Feature Models are generally preferred for those phases where there is an need to have a complete view of the whole system in order to handle the variability while Decision Models can be used when the whole system view is not needed and the SPL manager needs just to concentrate in the variants, because the commonalities are implicit. CVL is nearest to the physical models due to its direct link to DSLs and therefore it is more suitable to deal with actual physical components.

In order to address variability among engineering phases Feature Models or Decision Models can be used, since they can exploit their generic nature to easily manage the dependencies across different phases and with different involved engineers.

### 2.6.3 *CESAR Approach and Enhancements in PLE*

Within the CESAR project, a combination of several techniques in each of the analysis layer presented in Sect. 2.3.1 is proposed in order to solve specific PLE issues. This is depicted in Fig. 2.25 where it is shown the techniques that can be used at the same analysis level and those that can be used among different analysis levels.





**Fig. 2.25** Mapping of the variability approaches to engineering analysis phases

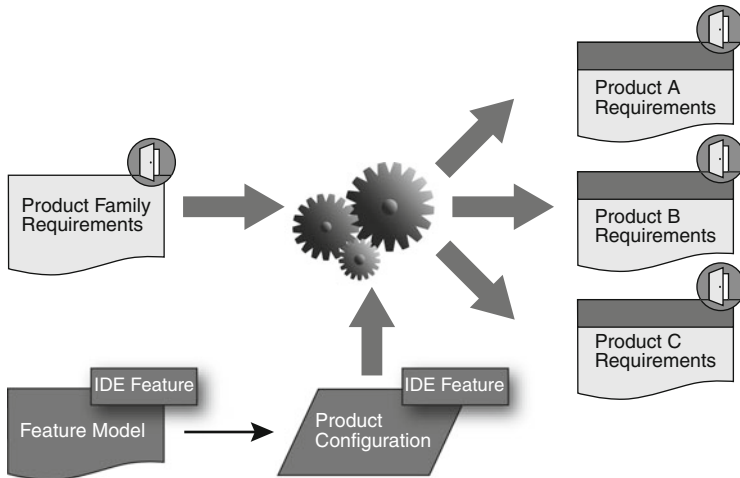
**Operation analysis level:** DM, thanks to its generic nature, is suitable to deal with business and customer needs and it is easier to see this variability needs as decisions. Moreover, the full description of the system is not really needed in order to take any decision on the variants, making DM the ideal solution for addressing variability. FM can also apply and a possible mixture of both could be interesting due to their different representational capabilities over a domain.

**Functional and non-functional need analysis level:** In this phase the engineer usually needs the whole view of the system in order to solve the variability, making FM the best choice. Some parts of the variability management can also be addressed by DM in order to speed up the global performance of model management (concentrating only in the variable parts). In this particular phase, the selection of the approach to be applied will largely depend on the exact type of system and the type of variability.

**Logical Architecture analysis level:** At this level, the variability can be tackled with all three options. However, the use of decision modelling will simplify and hide complexity of the whole system, by not showing what is actually not needed in order to solve the variability. The power of DM at this level lies in the ability to express in a more human intuitive way variability through questions and choices.

**Physical Architecture analysis Level:** This level is usually tight to a specific domain and, probably, to a specific DSL. Although the three options can be used, CVL seems the most suitable providing the mechanisms to solve the variability closer to the specific DSL.

In order to support variability management along the design process the CESAR RTP platform includes tools that can be used for Feature Modelling, Decision Modelling and the Common Variability Language approach. These tools are CVM (Compositional Variability Management) [34], PLUM (Product Line Unifier



**Fig. 2.26** Feature modelling approach for product family management

Modeller) [153] and CVL tool respectively. Appropriate tool-chains have been defined to support enhancements during PLE that aim to:

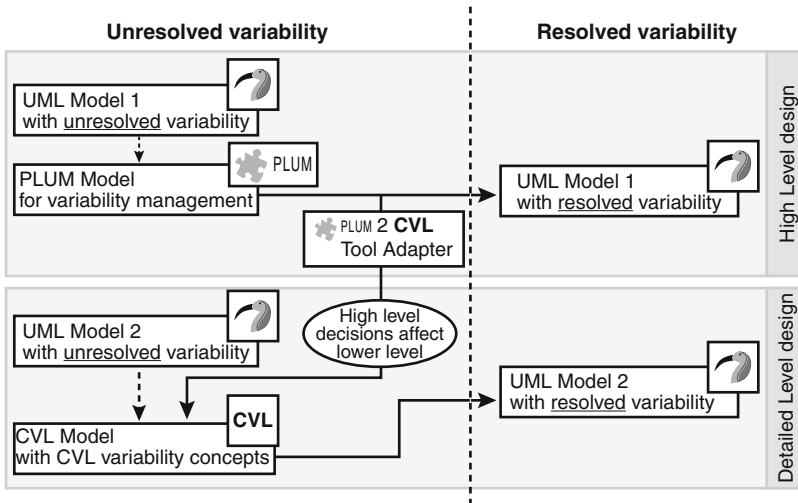
- Address complexity of handling variations points in complex system through the use of features models.
- Automate the transition from domain models (containing unsolved variation points) to resolution models

In the first case, each variability point of a product family is modelled in a feature model. Then product family requirements are developed and allocated to features and a subsequent configuration of the feature model identifies requirements for a specific product. The main idea of the corresponding tool-chain is to separate product configuration from requirements management as depicted in Fig. 2.26.

In the second case, system design models (defined, for example, using the Papyrus tool [125] in EAST-ADL [42] or other UML-like language) after being extended with variability data, they are processed by variability management tools like PLUM or CVL so that variation points are solved and specific resolutions models are generated. Both high level and detailed design can be addressed. The whole concepts is depicted in Fig. 2.27.

## 2.7 Traceability During Design Phases

Before closing the chapter related to the CESAR system life cycle, we should also say a few words related to traceability during design since it is considered a key element for any structured process development and a major prerequisite for critical system design. Much of the overall system certification credit is based on the



**Fig. 2.27** Example tool chain for automating resolution model generation

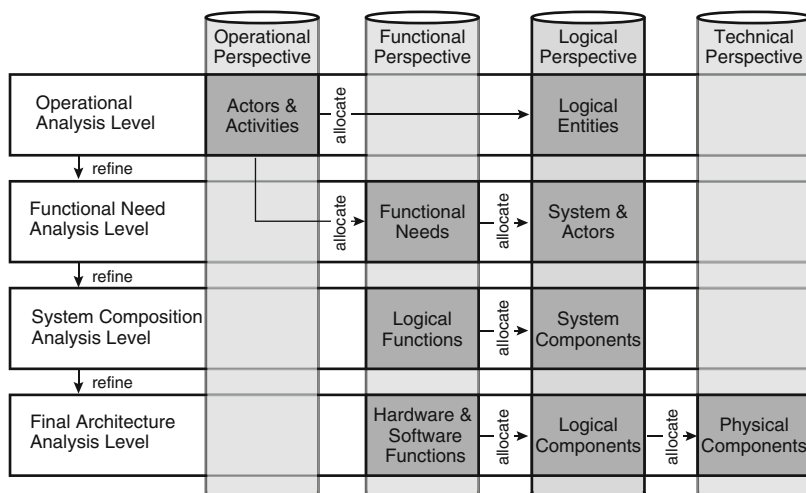
ability to demonstrate efficient traceability management between the artefacts of a system developed during the various design and development phases whether these concerns requirements, models, test cases, specifications, components etc.

The notion of traceability can be considered primary related to requirements as they are refined, satisfied or verified within the system life cycle, while we move from high level architecture analysis to detailed system specification. This is supported in CESAR since the envisaged CESAR common meta-model (CMM) foresees concepts like the “refine” link, which indicates that a new set of requirements can safely be substituted for the original set, or “satisfy” – “verify” links that express conformance of a design to a (set of) requirement(s), either because the designer asserted so or because V&V actions have substantiated it. Other requirement related link types are also possible. This part of traceability, related mainly to requirements, is described in more details in Chap. 3 (see Sect. 3.5.1).

Another form of traceability can be considered related to PLE. There it is usually necessary to establish clear traceability links between the configuration possibilities shown in the variability specification and the different variation points arising in the development process. This is considered part of product line management and is usually achieved through special PLE tools like PLUM.

A third form of traceability, of great interest during system design and especially in what concerns architecture analysis and iterative system refinement, will be briefly outlined in this section. It addresses the need to:

- Link models of the same system representing different aspects,
- Link elements referring to different types of analysis (i.e. logical architecture analysis elements with physical architecture elements) at the same abstraction level,



**Fig. 2.28** Elements and their relationships during system analysis

- Link elements referring to same or different types of analysis at different abstraction levels (usually in a top down analysis where hierarchical refinement applies)

In CESAR's design life cycle, it has been identified that these links may need to be established (a) between elements residing in the same model description file as well as (b) between elements in different models even when defined and developed using different tools.

The intra-model design traceability of (a) is supported through the CMM special "mapping" meta-class. This mapping can be used to describe the mapping solution between two elements which may belong to analysis models that are defined in different perspectives (of the same abstraction level) or in different abstraction levels. An "allocation" link is used in the former case while the link is described as "realization" in the latter case. Figure 2.28 illustrates the "allocate" relationships that are presumed to be defined between the most commonly used perspectives during the various system analysis levels as presented in Sect. 2.3.1. The examples presented in Chap. 5 (DOORS Management system) may provide more concrete insight on how traceability is enabled between the perspectives.

To address inter-model design traceability between elements defined in different model files, a necessity when designing large systems where many different teams may be involved, CESAR has identified the need for a lightweight data integration layer that will enable to manage the creation/editing of links between any models and/or internal model elements stored in a model repository. This has become possible in the RTP by the development of a special linking service, which provides uniform access to models and their internal elements in a manner that is independent on how these models are defined and implemented, and the existence

of an appropriate repository that can be used to store the defined links. Through this link repository service, different kinds of link, such as “satisfy”, “refine”, “implement” and so on can be defined and managed (actually the exact types of links supported is left open and may vary depending on each domain needs). More details on the link repository mechanism can be obtained from Chap. 6.

The two approaches described above are completely independent one from the other. The inter-model one is more generic since it may address elements of any kind developed in completely different tools. However, there is no prohibiting factor in using both of them in a complementary function.

CESAR - Cost-efficient Methods and Processes for  
Safety-relevant Embedded Systems

Rajan, A.; Wahl, Th. (Eds.)

2013, XIV, 391 p., Hardcover

ISBN: 978-3-7091-1386-8