# Chapter 5
# Component-Based Development

**Kung-Kiu Lau, Marc Pantel, DeJiu Chen, Magnus Persson, Martin Törngren, and Cuong Tran**

In this chapter, we focus on the use of component-based development (CDB) in CESAR. First, we introduce what we mean by components and how they fit into the product lifecycle. Then, we report on the two major technological innovations of the project: the X-MAN and HRC frameworks.

## 5.1 What Is Component-Based Development?

The aim of component-based development is to (i) build components and deposit them in a repository; and (ii) use or reuse these pre-existing components to build many different systems by assembling the components using pre-defined composition mechanisms. This is illustrated by Fig. 5.1.

For a given problem domain, components are identified and developed, using domain knowledge. Similarly, the composition mechanisms for the components are defined and fixed for the domain.

For hardware systems, component-based development is of course standard practice, and components are standard parts that can be assembled in pre-specified ways. For instance, chips can be assembled by wiring their pins together.

For software systems, the most widely accepted generic component is depicted in Fig. 5.2: it is a software unit with *provided services* (lollipops) and *required services* (sockets). These components are composed by connecting provided services with matching required ones, as shown in Fig. 5.3.

D. Chen · M. Pantel · M. Persson · M. Törngren
Kungliga Tekniska Högskolan, Stockholm, Sweden

K.-K. Lau (✉) · C. Tran
The University of Manchester, Manchester, United Kingdom
e-mail: kung-kiu@cs.man.ac.uk

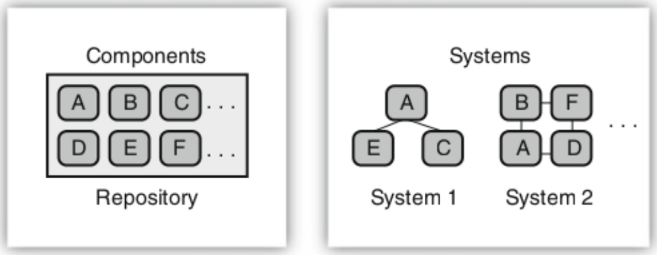


**Fig. 5.1** Component-based development

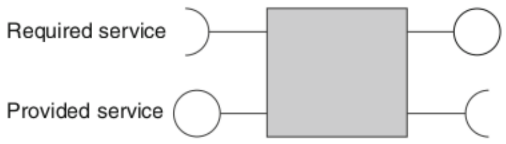


**Fig. 5.2** A generic component

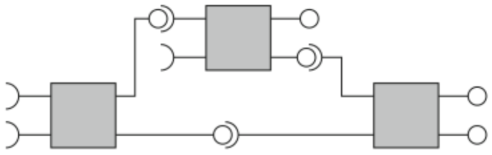


**Fig. 5.3** Composing generic components

Different types of components are defined as different instances of the generic component, together with corresponding composition mechanisms. Such definitions are given by *component models*.

## 5.2 Component Models

A *component model* [77, 105] defines: (i) what components are; and (ii) composition mechanisms for composing the defined components.

For software systems, there are three categories of component models [105] with (i) *objects*; (ii) *architectural units*; and (iii) *encapsulated components* as components, respectively. Table 5.1 shows these categories and examples of component models therein. The three kinds of components are illustrated in Fig. 5.4.

An object is similar to a generic component except that an object does not have or show any required services (hence sockets in dotted lines). An architectural unit is just the same as a generic component, except that an architectural unit uses ports instead of services. An encapsulated component has only provided services, but no required services; it therefore performs all its computation within itself, i.e., it does not call another (encapsulated) component.

**Table 5.1** Categories of component models

| Category | Components | Sample component models |
|---|---|---|
| (i) | Objects | EJB |
| (ii) | Architectural units | CMM, EAST-ADL, AADL |
| (iii) | Encapsulated components | X-MAN |

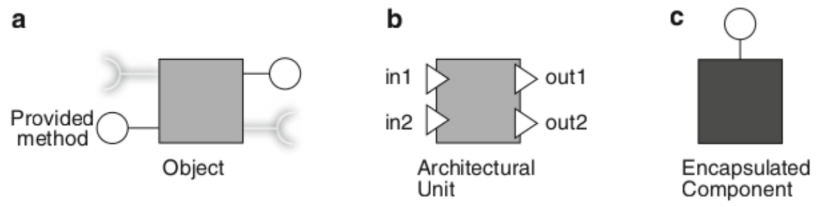


**Fig. 5.4** Types of components

**Table 5.2** Composition mechanisms

| Category | Components | Composition mechanism |
|---|---|---|
| (i) | Objects | Method delegation |
| (ii) | Architectural units | Port connection |
| (iii) | Encapsulated components | Coordination |

The composition mechanisms for these three kinds of components are shown in Table 5.2.

Objects "compose" by method delegation, i.e., direct method calls (or direct message passing). Architectural units compose by connecting their ports (or indirect message passing). Encapsulated components compose by coordination, i.e., exogenous connectors that coordinate control flow between components. These composition mechanisms are illustrated in Fig. 5.5.

For software systems, a number of development processes, or *life cycles*, for CBD have been proposed, e.g. [27, 29, 37, 100, 142], to name a few. (A recent survey can be found in [98].) Naturally these processes all reflect the desiderata of CBD [23], and converge on the general view depicted in Fig. 5.6.

The generic CBD process in Fig. 5.6 comprises two separate processes: one for *component development*, and one for component-based *system development*. Component development is also known as "development *for* reuse", since it is concerned with developing components that can be stored in a repository and (re)used to build different systems. Component-based system development is also known as "development *with* reuse", since it is concerned with developing systems by reusing pre-built components (the result of the component development process).

Each process beginning follows the same life cycle of "requirements analysis, design, implementation, testing and maintenance". For component development, implementation is a single activity, whereas for system development,

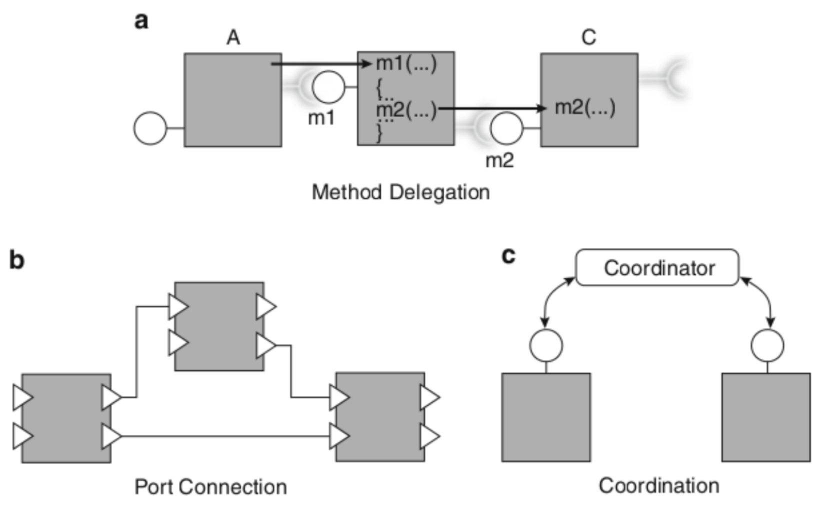


**Fig. 5.5** Composition mechanisms

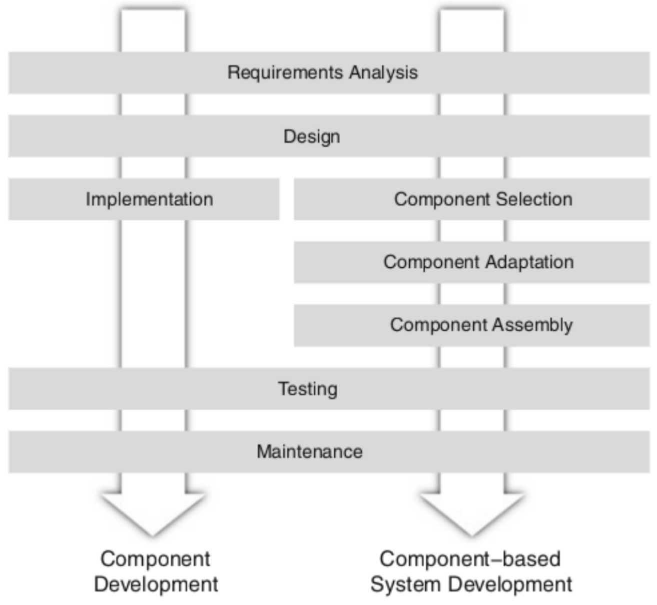


**Fig. 5.6** Standard CBD processes

implementation is a sequence of activities based on pre-built components, namely component selection, adaptation and assembly.

Figure 5.6 does not differentiate between the context for the component life cycle and that of the system life cycle. In fact, the component life cycle is followed

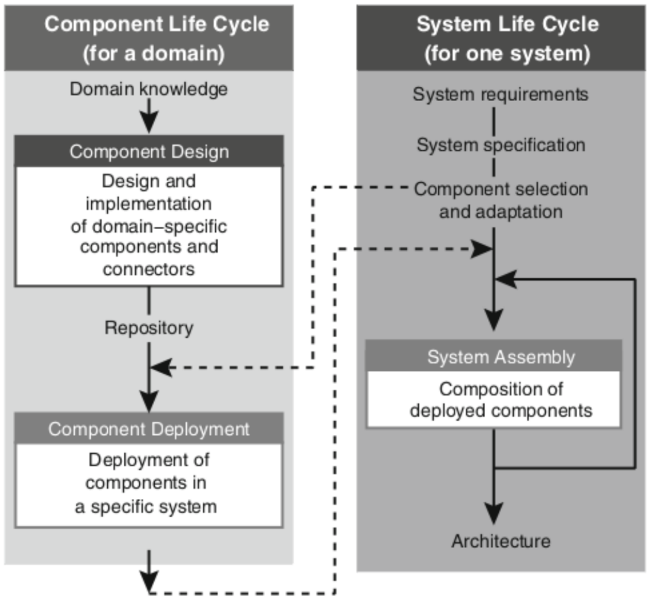

Component Life Cycle (for a domain)
Domain knowledge
Component Design
Design and implementation of domain–specific components and connectors
Repository
Component Deployment
Deployment of components in a specific system
System Life Cycle (for one system)
System requirements
System specification
Component selection and adaptation
System Assembly
Composition of deployed components
Architecture

**Fig. 5.7** Idealised CBD life cycle

by the component developer for the whole domain, whilst the system life cycle is followed by the system developer for each system to be constructed. This distinction is illustrated in Fig. 5.7.

Figure 5.7 differs from the standard CBD process (Fig. 5.6), in that its component life cycle is a more complete one, namely the idealised one [105]. The idealised component life cycle is called so because it meets all the desiderata of CBD that have been identified in the literature [23]. It consists of two phases: component *design* and component *deployment*, and is set in the context of a problem domain. In the design phase, components are (identified and) designed and constructed according to the domain requirements or knowledge [106], and deposited into a *repository*. Repository components are domain-specific but not system-specific. In the deployment phase, components are retrieved from the repository and instantiated into executable component instances which are then deployed into a specific system under construction.

The system life cycle also differs slightly from that in Fig. 5.6 in that system design is now replaced by a completely bottom-up process of component *selection* (from the repository) and *adaptation*, followed by (component *deployment* in the component life cycle followed by) *system assembly*, which is simply the composition of the deployed components. The bottom-up nature of this process is indicated by an iterative loop in Fig. 5.10. It is worth noting that within this loop, the component life cycle links up with the system life cycle, since deployed components (from the component life cycle) are iteratively assembled into the system under

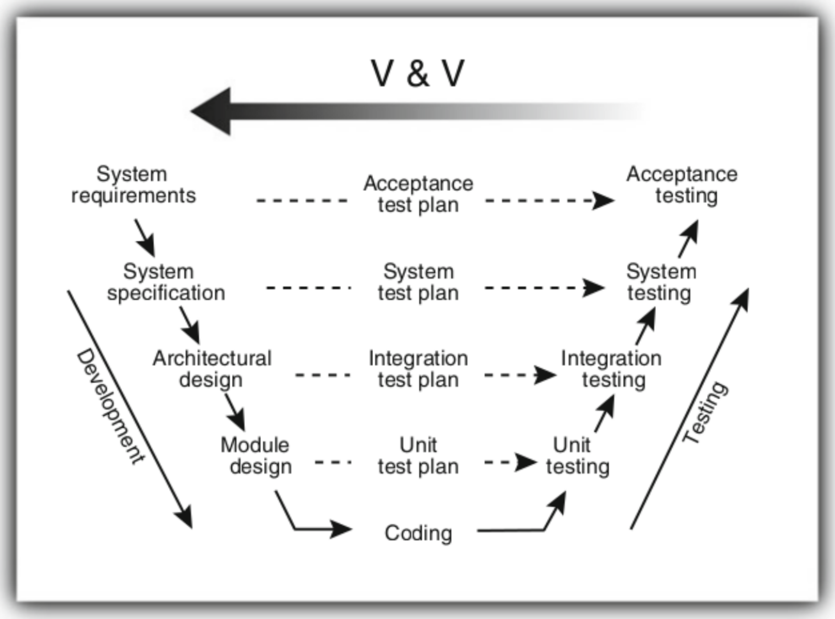


**Fig. 5.8** The V-Model

construction (in the system life cycle). This link is denoted by the dotted arrows between the two life cycles in Fig. 5.7, via the step of component selection and adaptation, and the step of component deployment.

### 5.2.1 The W-Model

The CBD process in Fig. 5.6 does not explicitly address Verification and Validation (V&V). For general (modular) system development, the standard model for V&V is the V-Model [159]. The V-Model is an adaptation of the traditional waterfall model for *modular* system development. It defines a process consisting of phases for requirements, system specification, system or architectural design, module design, implementation and testing. Implementation consists of coding for the individual modules, and coding for integrating the modules into the entire system using the architectural design for the system. Testing follows coding [159] (Fig. 5.8). Thus, the coding phase divides the whole process into *development*, the left arm of the V, and *testing*, the right arm of the V.

The key property of the V-Model that is pertinent here is that it is a *top-down* approach to system design and development, as Fig. 5.8 shows. First, a top-level design is made of the architecture of the entire system; this identifies and specifies subsystems or modules, and their inter-relationships. Then the individual modules are designed according to their specifications in the top-level design. In general, this top-down approach may be applied successively, each time decomposing subsystems or modules in the current level of design into further sub-systems or modules. This decomposition is repeated as many times as is necessary, until a final design is arrived at in which the design of the system as well as all the individual modules is deemed complete, i.e., no further decomposition is necessary or desirable.

Compared to the standard CBD process in Fig. 5.6 and the idealised CBD life cycle in Fig. 5.7, both of which contain two life cycles,[1] one for component development and one for system development, the V-Model contains only one life cycle, for system development.

Furthermore, the standard CBD process in Fig. 5.6 and the idealised CBD life cycle in Fig. 5.7 also show CBD as an essentially *bottom-up* approach to system design, in the sense that components have to be developed first (in the component life cycle), and any particular system is constructed from these components (in the system life cycle). In contrast, the V-Model (Fig. 5.8) is essentially a *top-down* approach to system design: the system is designed first (thus identifying the required components), and then components are developed.

A straightforward adaptation of the V-Model for CBD would be to retain the top-down approach to system design but use a component as a module, as shown in Fig. 5.9.

For example, the V-Model adopted by the avionics industry as a CBD process (e.g., Airbus processes [64, 67]) is such an adaptation.

However, such a straightforward adaptation of the V-Model is at variance with the standard CBD process in Fig. 5.6, precisely because it does not include a component life cycle and consequently does not incorporate the bottom-up nature of CBD.

An adaptation of the V-Model for CBD that does incorporate the bottom-up nature of CBD is that of [36]. It does so by containing separate life cycles for component development and system development, like in Fig. 5.6. However, this adaptation really applies the V-Model only to its system life cycle; there is no evidence of the V-Model in its component life cycle (which is the same as the one in Fig. 5.6).

In our view, to adapt the V-Model properly for CBD, we need not only to incorporate both the component life cycle and the system life cycle, but also to apply the V-Model to *both* of these cycles. In addition we need to specify a component model that defines the components (and their composition) properly. (A definition and survey of component models can be found in [105].)

[1]We will use ‘life cycle’ interchangeably with ‘development process’ or simply ‘process’.

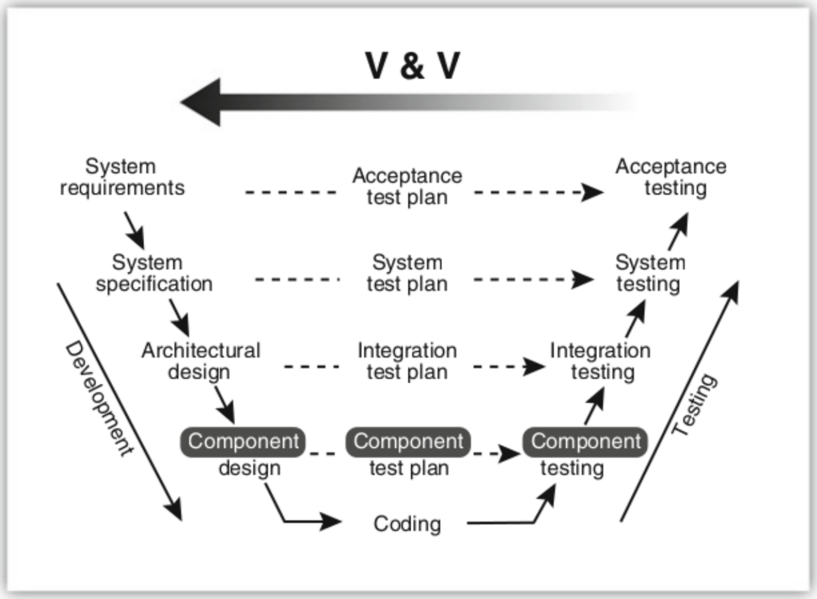


**Fig. 5.9** Adapting the V-Model for CBD

We have defined such an adaptation, using a component model that we have defined ourselves. Now we describe our adaptation, which we call the W-Model,[2] for reasons that will become apparent later.

In CESAR, we use the X-MAN component model [107, 108, 169],[3] and we have defined a CBD process based on X-MAN. This process is the one shown in Fig. 5.7.

Applying the V-Model to both the component and system life cycles yields a CBD process with V&V as shown in Fig. 5.10.

Compared to the straightforward adaptation of the V-Model in Fig. 5.9, *component V&V* (which corresponds to component testing in Fig. 5.9) now occurs in the component life cycle, whilst *compositional V&V* (which corresponds to integration testing in Fig. 5.9) and *system V&V* (which corresponds to system testing in Fig. 5.9) occur in the system life cycle.

The X-MAN CBD process with V&V in Fig. 5.10 can be re-cast straightforwardly as a process with two conjoined V-Models, one for the component life cycle and one for the system lifecycle. These two V-Models are conjoined via the step of component selection, adaptation, and deployment. This "double V" process is

[2]The name W-Model has also been used in software testing [146] and product line engineering [112] in the context of traditional (i.e., non-CBD) software engineering.

[3]This name was not used in these papers, but has been created within the CESAR project.

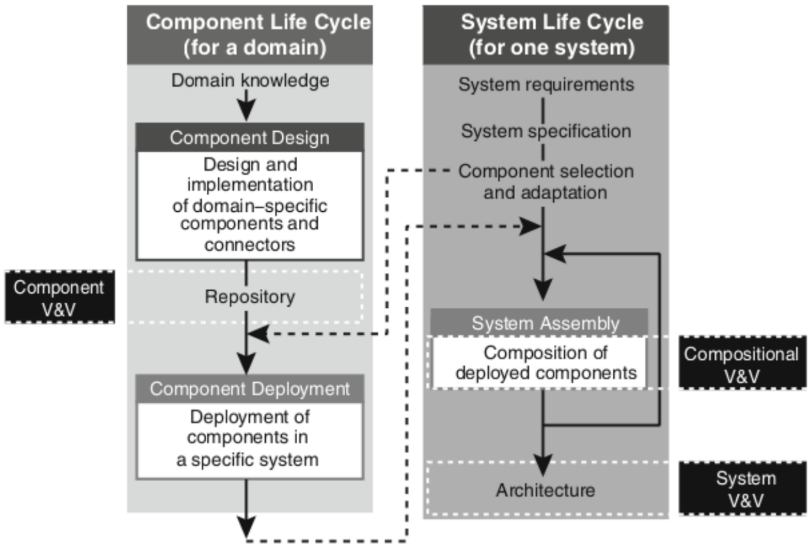

Component Life Cycle (for a domain)
Domain knowledge
Component Design
Design and implementation of domain–specific components and connectors
Repository
Component Deployment
Deployment of components in a specific system
Component V&V
System Life Cycle (for one system)
System requirements
System specification
Component selection and adaptation
System Assembly
Composition of deployed components
Architecture
Compositional V&V
System V&V

**Fig. 5.10** X-MAN CBD process with V&V

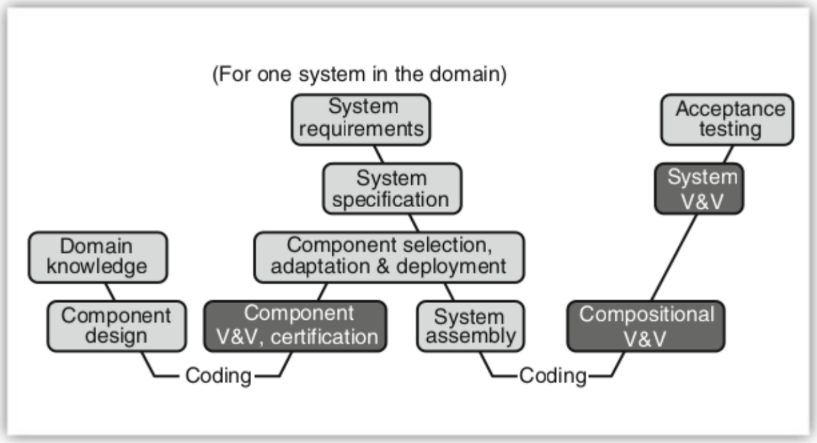


**Fig. 5.11** The W-Model

shown in Fig. 5.11. We therefore called it the *W-Model* [109].[4] We have highlighted the V&V activities in the W-Model by boxes with black borders.

[4]In English, "W" is pronounced "double u"; there is no letter pronounced "double v".

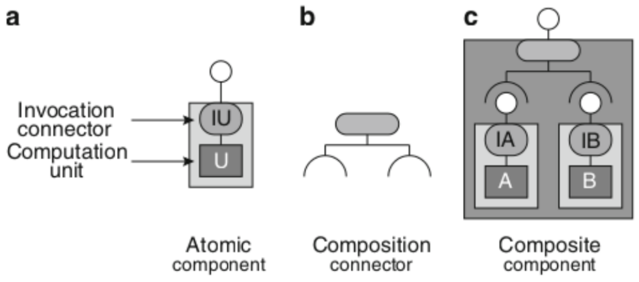


**Fig. 5.12** X-MAN component model

## 5.3 Implementing CBD with X-MAN

In the CESAR project, CBD is implemented in the form of a tool for X-MAN. This tool implements the X-MAN component model as well as the W-Model. The X-MAN tool is implemented in a model-driven manner. It has been implemented using the GME toolkit [69]. In GME, meta-models that contain definitions of elements, structures and syntax first have to be defined using a class diagram notation similar to UML class diagrams. Models can then be created by instantiating the pre-defined meta-models. To provide behaviours for models, GME allows us to develop interpreters that can interact with models, i.e., execute or manipulate models.

To implement the W-Model, we had to implement: (i) the X-MAN component model, for defining and constructing components and their composition mechanisms; (ii) the component life cycle in the W-Model; (iii) the system life cycle in the W-Model; (iv) the link between the component and system life cycles; (v) component V&V; (vi) compositional V&V; and (vii) system V&V. For lack of space, we cannot describe all these fully; so we only highlight the key elements of the implementation, and briefly describe them.

### 5.3.1 The X-MAN Component Model

The X-MAN component model has been described in several papers, e.g., [107,108, 169]. Here we give a brief summary.

In X-MAN there are two kinds of components: atomic and composite (Fig. 5.12). An atomic component contains a computation unit and an invocation connector. The computation unit provides methods or functions which can be invoked via the invocation connector. An atomic component is encapsulated in the sense that all its computation occurs within its computation unit, i.e., an atomic component does not invoke the methods of other components. Thus an atomic component has only provided services (denoted by lollipops) but no required services.

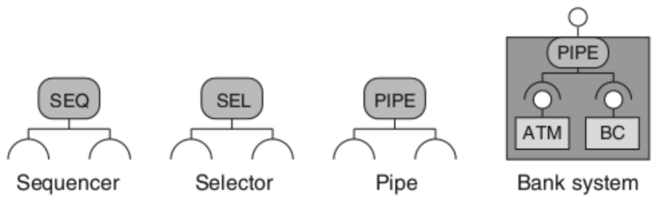


**Fig. 5.13** X-MAN composition connectors

A composite component (Fig. 5.12c) is built from atomic components by composition connectors (Fig. 5.12b). A composition connector coordinates control (as well as data flow) between components, i.e., it coordinates the invocation of services between components. A composite component is also encapsulated, and it also has only provided services; this is a direct consequence of the encapsulation of an atomic component. In general, composite components are self-similar; this is important as it means that we have *compositionality*. In particular, this enables us to do *compositional V&V*.

The X-MAN connectors include the Turing-complete set of control structures: sequencing, branching and looping. Sequencing and branching are composition connectors for multiple components, whereas looping is an adaptor for a single component. For sequencing, we have the *sequencer* and *pipe* composition connectors; for branching, we have the *selector* composition connector (Fig. 5.13). A sequencer only passes control to the next component, whereas a pipe passes control and results from the first component to the next component, as shown in the bank system example in Fig. 5.13.

In the X-MAN Tool, the X-MAN component model is implemented as part of the component life cycle and the system life cycle (see below).

## 5.3.2 The Component Life Cycle

As mentioned in Sect. 5.2.1, the component life cycle in the W-Model is the idealised one, consisting of the component design phase and the component deployment phase. To implement the component life cycle in GME, we first need to define meta-models for these phases. However, as we also pointed out in Sect. 5.2.1, the component life cycle links up with the system life cycle during component deployment and system assembly. Therefore, it is more convenient to put the component deployment phase in the meta-model for the system life cycle. Consequently, the meta-model of the component life cycle consists of just the meta-model for component design phase.

To implement the component life cycle in GME, we define the meta-model for the component design phase and implement an interpreter for it. We also implemented a component repository including interpreters to support component deposit and retrieval.

#### 5.3.2.1 The Meta-model

The meta-model for the component life cycle is presented in Fig. 5.14. It defines component design according to the X-MAN component model.

The top-level element is a *Design*, which contains the definition of a (atomic or composite) component according to X-MAN (Figs. 5.12 and 5.13). The property *ExecutableCode* of a *computation unit* provides computation implementation. Each *method* in a computation unit can be augmented with a contract that is defined by pre- and post-conditions. This contract is used for *component V&V*.

Each atomic or composite component must have one *interface* that consists of one or more *service*s and *data element*s. A *service* contains references to methods provided by the component's computation unit. A *data element* denotes a datum that is required or provided by the component. As components in the design phase are templates, data elements are characterised with type, min, max and default values; actual values are provided when instantiating these templates during component deployment. Finally, a component can have an invariant, which is defined in its interface.

#### 5.3.2.2 Component Designer

We have constructed a Component Designer that supports component design, V&V, and storage. Figure 5.15 shows a component, namely *Locker*, under construction in the Designer, with a design palette on the left, tree view of the design on the right, and the main design view in the middle. The main view shows that *Locker* is an atomic component built from connecting an invocation connector *Locker_INV* to a computation unit *Locker_CU*. The component exposes its interface containing the provided service *getLocking*.

When completed, a component can be stored into the component repository. This activity is triggered by selecting the *DEP* button on the toolbar (see close-up in Fig 5.15).

#### 5.3.2.3 The Repository

For the component repository, we used the Firebird database [62] and for database access we used the SQLAPI driver [147].

### 5.3.3 The System Life Cycle

The system life cycle contains the iterative loop of component selection and adaptation, component deployment and system assembly. To implement this life cycle in GME, we defined a meta-model for component deployment and system

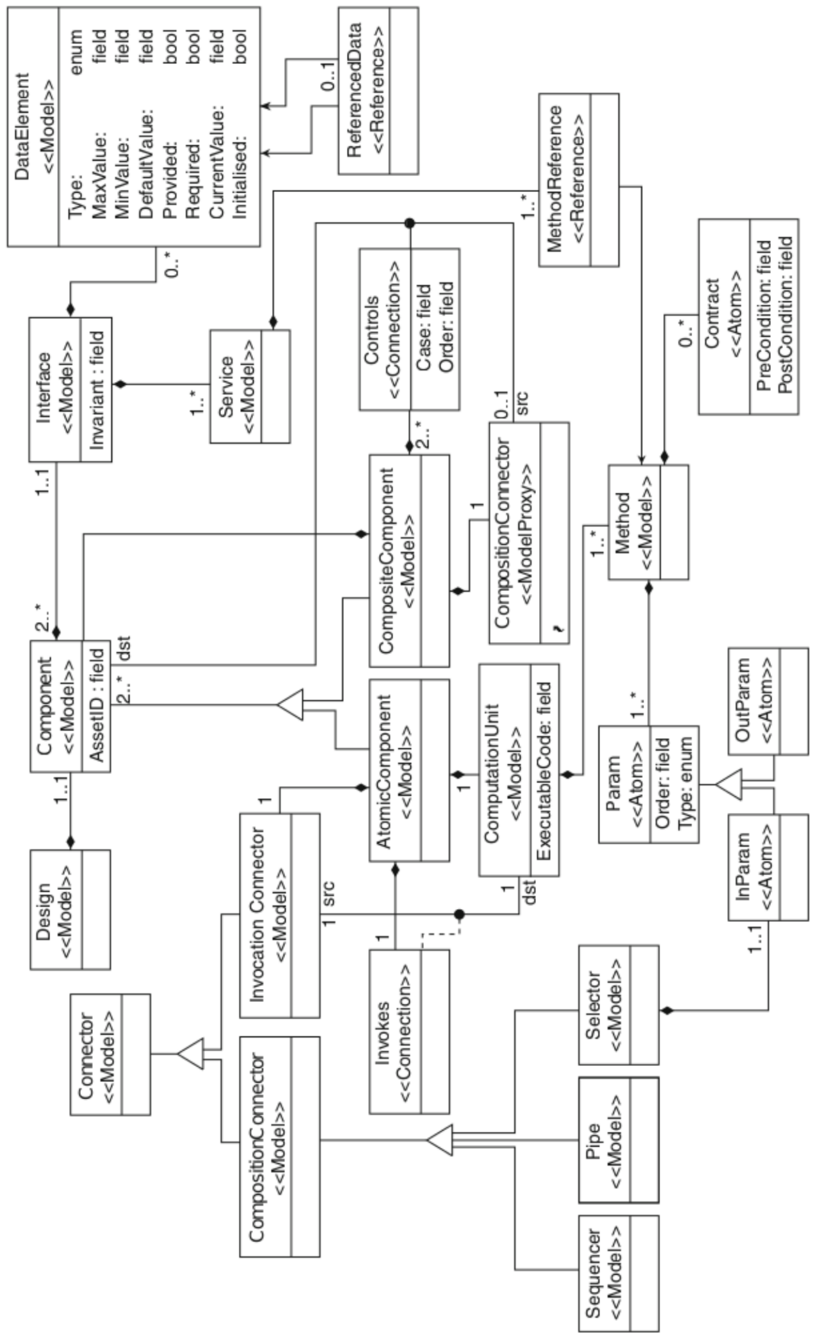


**Fig. 5.14** Meta-model for the component life cycle

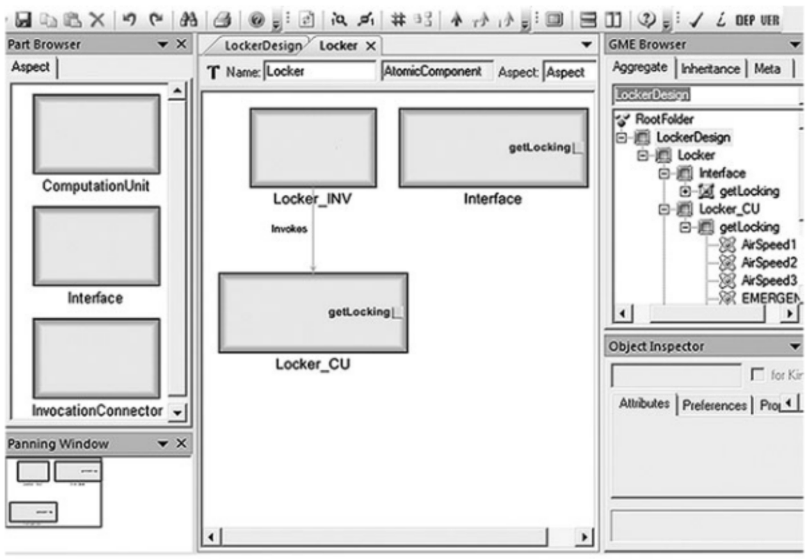


**Fig. 5.15** Component Designer

assembly together, and provide an interpreter for it. This interpreter also provides for component adaptation. Component selection makes use of the retrieval function implemented for the repository in the component life cycle.

Systems constructed in the system life cycle are executable. To provide for system execution, we developed a system simulator in the form of another GME interpreter.

#### 5.3.3.1 The Meta-model

The meta-model of the system life cycle is depicted in Fig. 5.16. It defines system assembly from (deployed) component instances according to the X-MAN component model.

The top element is a *System*, as opposed to (*Component*) *Design* in the component life cycle. Like a component, a *system* has one *system interface* that contains some services and data elements. A *system reference* denotes a reference to another system. Sub-systems, like component instances, can be composed using composition connectors. Therefore, systems can be constructed incrementally from (sub-)systems. Finally, like components in the component life cycle, component instances can have invariants and methods can have contracts. These are used for *compositional V&V*.

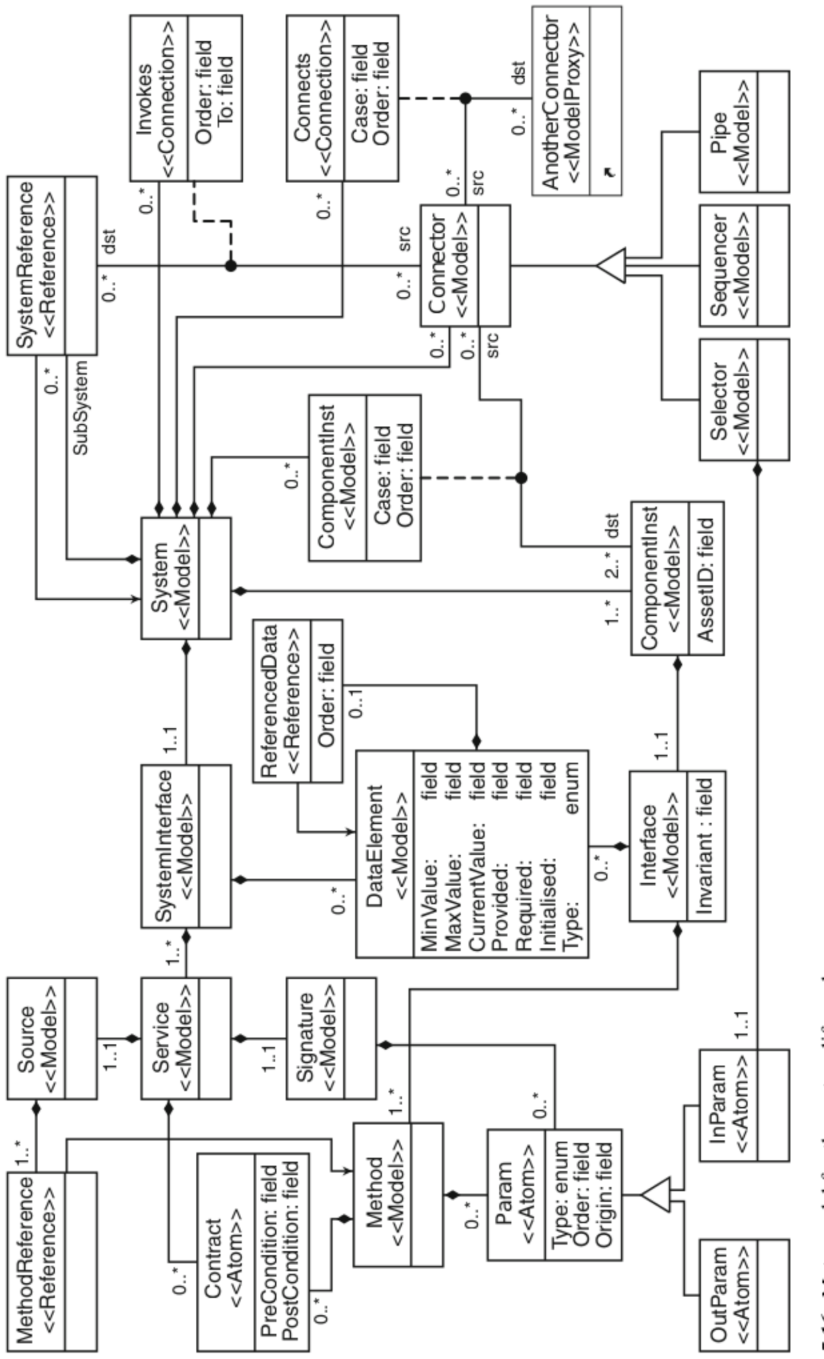
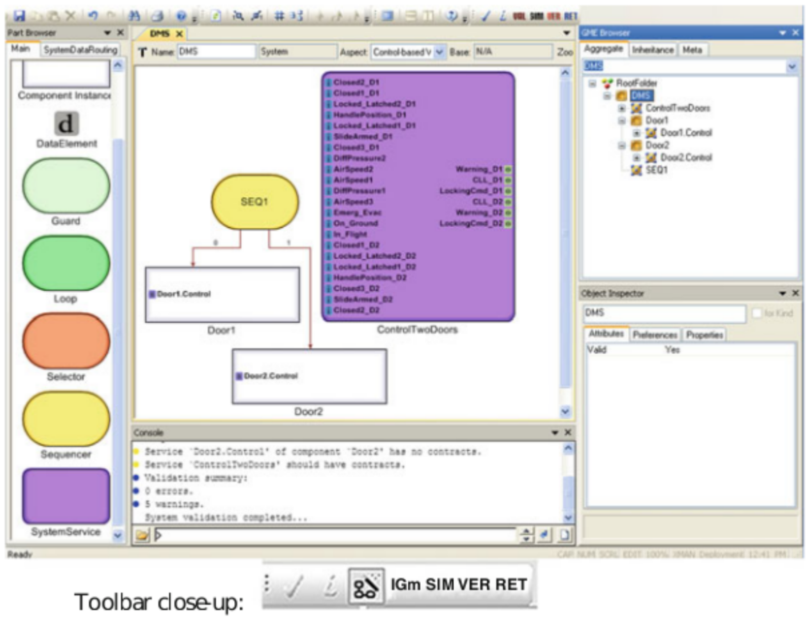


**Fig. 5.16** Meta-model for the system life cycle



```
Service 'Door2.Control' of component 'Door2' has no contracts.
Service 'ControlTwoDoors' should have contracts.
Validation summary:
0 errors.
5 warnings.
System validation completed...
```

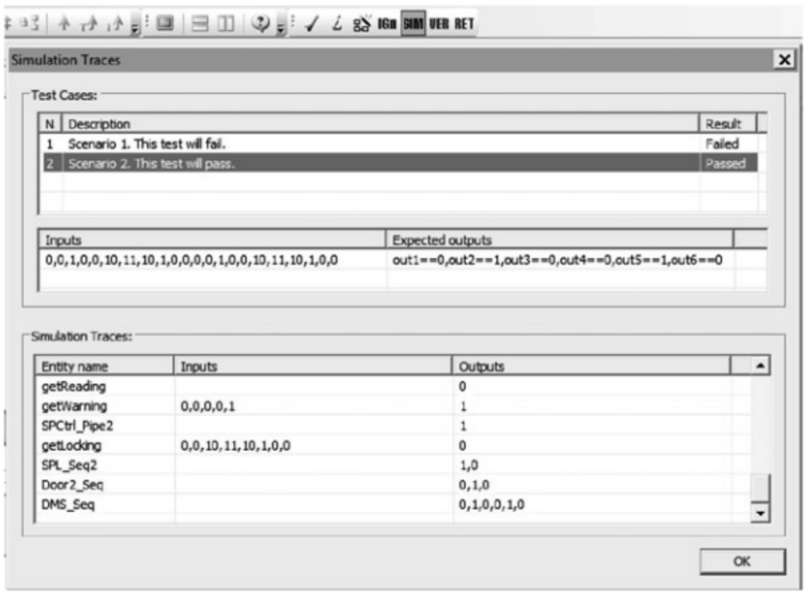
**Fig. 5.17** System Assembler

### 5.3.3.2 System Assembler

We have constructed a System Assembler for composing selected deployed component instances. Components are selected from the repository and instantiated, and the instances are then deployed. Component instance composition, i.e., system assembly, to create the system follows after that. Similar to the Component Designer, the System Assembler offers a design palette, a tree view and a main view for system design. It is illustrated in Fig. 5.17 in which the developed system is built from component instances Door1 and Door2, using connector SEQ1.

### 5.3.3.3 The Simulator

The simulator asks a developer to select a provided service and supply the required inputs. It then traverses the system architecture and provides behaviour to our connectors (Pipe, Sequencer, Selector). Furthermore, when reaching a component instance, the simulator has to retrieve the component design (from the component life cycle) and traverses its structure with deployment data until reaching a computation unit to execute its computational source code to perform the desired computation. The simulator passes through the system and finally returns to the top-level connector with some outputs. Moreover, our simulator can take test cases that specify inputs and expected outputs, that are defined as simple assertions, e.g.



| N | Description | Result |
|---|---|---|
| 1 | Scenario 1. This test will fail. | Failed |
| 2 | Scenario 2. This test will pass. | Passed |

| Inputs | Expected outputs |
|---|---|
| 0,0,1,0,0,10,11,10,1,0,0,0,0,1,0,0,10,11,10,1,0,0 | out1==0,out2==1,out3==0,out4==0,out5==1,out6==0 |

| Entity name | Inputs | Outputs |
|---|---|---|
| getReading | | 0 |
| getWarning | 0,0,0,0,1 | 1 |
| SPCtrl_Pipe2 | | 1 |
| getLocking | 0,0,10,11,10,1,0,0 | 0 |
| SPL_Seq2 | | 1,0 |
| Door2_Seq | | 0,1,0 |
| DMS_Seq | | 0,1,0,0,1,0 |

**Fig. 5.18** Simulation results

*output1 equals 1*. The simulator then evaluates these assertions with system outputs and additionally provides simulation traces to support system execution analysis.

To implement the simulator, we assumed that computation unit implementation is given in the C programming language. We then used C++ to implement our connector behaviour and the interpreter itself. Moreover, to execute computation unit C code, we adopted the Ch library [158] to interpret without the need of compilation.

Figure 5.18 shows the screen after the simulation completes successfully. The screen depicts the two test cases description and the result. As we can see, the first test failed and the second test passed. When a test case is selected, the screen shows its specified input and output assertions. In addition, simulation traces are displayed to support further analysis.

## 5.3.4 Validation & Verification

All V&V activities rely on the CBMC model checker [33], and are carried out by a separate tool that is interfaced to the X-MAN tool. For activating this tool, the X-MAN Tool provides the *VER* button on the toolbars of the Component Designer and System Assembler (see close-up in Fig. 5.15). The X-MAN Tool exports the component design or system assembly into an exchangeable format, e.g., XML, and passes the design to the model checker.

As shown in Figs. 5.10 and 5.11, there are three V&V activities: *component, compositional, and system* V&V. Component V&V is for components when they are designed and built in the component life cycle, using the Component Designer. To carry out component V&V, the developer presses the *VER* button on the toolbar (Fig. 5.15). This V&V is carried out by the verification tool, and is with respect to the contracts of the component. After component V&V, components are deposited in the repository. Repository components can thereafter be certified (Fig. 5.11).

Compositional V&V is carried out during system assembly in the system life cycle: for any iteration of the system assembly process, V&V of the current system can be carried out with respect to the contracts of the system. These contracts are composed from the contracts of the (sub)components. Compositional V&V thus reflects the bottom-up nature of CBD: component V&V occurs first, followed by V&V of composites built from the components. It also reflects an important property of composition in the X-MAN component model, namely *compositionality*, which means that properties of a composite can be derived (via composition) from those of its sub-components. This provides a means of re-using the V&V of the sub-components.

When the system is finally completed, it may require further V&V to check for additional system properties that may not be compositional, such as emergent properties. The model checker can check such properties.

For compositional and system V&V, the developer presses the *VER* button on the toolbar of the System Assembler (Fig. 5.17).

### 5.3.5 An Industrial Example: The Door Management System

Now we present an industrial example from the aerospace domain that we have implemented using the X-MAN Tool. The example is a simple version of the Door Management System (DMS) on a civil aircraft[5]; it manages only 2 passenger doors (the most complicated system can consist of up to 14 doors including passenger, cargo and emergency doors).

Structurally, each door is equipped with a number of sensors that produce status readings: “closed”, “lock-latched”, door handle position, and “slide armed”. In addition, there are inputs coming from other systems on the aircraft. The inputs are “on ground”, “in flight”, differential pressure, emergency and air speed signal readings.

Functionally, the system monitors the door status, manages the evacuation slide, and controls the flight lock actuator for each door. The door status is calculated from “closed”, “locked”, and “latched” signals read from two “closed” and three “lock-latch” sensors on every door. Also, a warning is activated when the inner door handle is moved and the evacuation slide is in the “armed” position. The evacuation

[5]This is one of the pilot applications in the CESAR project.

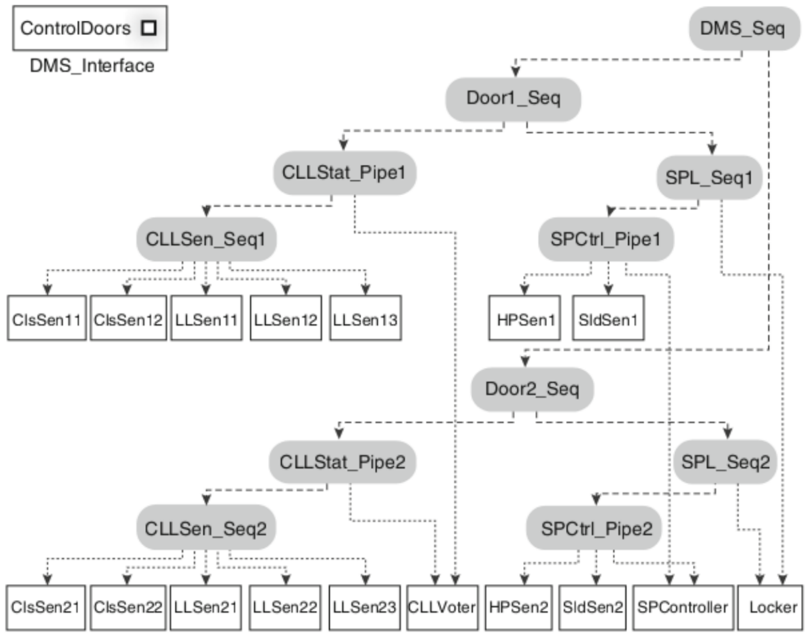


**Fig. 5.19** The aircraft doors management system as implemented in X-MAN

slide management furthermore generates a warning if the aircraft is "on ground" and the cabin is still pressurised, but the slide is disarmed. Finally, the flight lock control only permits the door to be unlocked if the aircraft is "on ground".

From our analysis of the DMS system, we identified four components: Sensor, CLLVoter, SlidePresWarnController, and Locker. These components are common and can be used many times in possible variants of DMS, e.g. a DMS variant consisting of four passenger doors. Hence, in the component life cycle, we designed the four components and deposited them in the component repository. The Sensor component offers the *getReading* service to get sensor reading; CLLVoter provides *vote* service that calculates door status; SlidePresWarnController offers *getWarning* service that manages the evacuation slide; and Locker has *getLocking* service to control the flight lock.

In the system life cycle, we implemented the DMS by using pre-defined components. The complete system is depicted in Fig. 5.19. In the architecture, rounded rectangles are composition connectors and normal rectangles are component instances. The rectangle on top is the system interface that specifies the provided service of DMS. Our implementation of DMS uses four pre-defined components from the component repository. The Sensor is instantiated 14 times with different port values to implement different sensors, i.e., ones that read "closed", "lock-latched", handle position, and "slide armed" values. The other three components, which are CLLVoter, SlidePresWarnController, and Locker, are instantiated once to make CLLVoter, SPController and Locker respectively.

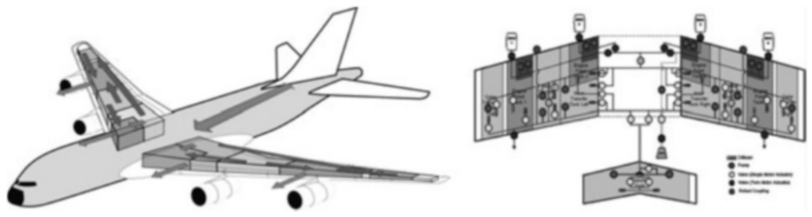


**Fig. 5.20** The fuel management components in an aircraft

The architecture consists of two clusters. Each cluster represents one door. We executed the DMS using our simulator. We built a test case file with two test cases in which inputs and expected outputs are specified. These simulation parameters can be seen in the middle table in Fig. 5.18. The simulation outputs returned by the top-level connector DMS_Seq are presented in the last table in Fig. 5.18. The outputs are 0, 1, 0, 0, 1, 0 which means the doors are not closed, lock-latched, evacuation slide warning is activated and the flight lock cannot be engaged. These values match the expected output specified in the test case, hence the test passed.

## 5.4 Case Study: The Ground Fuel Transfer Function

In [74] we presented a case study on a representative avionics application – the Ground Fuel Transfer function of a large transport aircraft. Ground fuel transfer is a specific function of the fuel management system of an aircraft. Figure 5.20 shows a schematic overview of the components involved in fuel management on modern systems, involving multiple tanks, and numerous pumps and valves connecting these. There are several functions to manage, including safety critical ones. These functions include refuel/defuel, wing bending relief, and communications to and from several other systems. Ground fuel transfer implements the specific behaviours of the fuel management system when the aircraft is physically on the ground, as opposed to behaviours while the aircraft is in flight.

### 5.4.1 The Component Structure

Recall that the behaviour of an X-MAN component is defined by means of a set of operations. The ground fuel transfer system consists of six selective top-level operations which are mutually exclusive (Fig. 5.21): *Automatic Refuel* (AR), *Manual Refuel* (MR), *Defuel* (DF), *Ground Transfer* (GT), *Shut-Off Test* (SOT) and *OFF*. Each of them is further composed of a set of sub-operations. The sub-modules are shown for the *Manual Refuel* operation only.

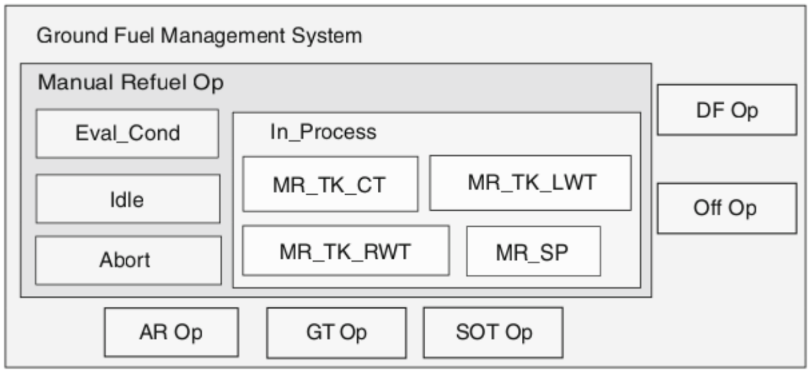


**Fig. 5.21** A functional composition overview of a ground fuel management system

As given in the largest box of Fig. 5.21, the *MR* operation is composed of four sub-operations: *Eval_Cond*, *Idle*, *In-Process*, and *Abort*. The last three sub-operations are also selectable on a mutually exclusive basis. *Eval_Cond* shall be called first to decide which of the rest three sub-operations is chosen to be executed next at the runtime. Furthermore, the *In-Process* consists of another four sub-components in sequence. They determine the respective operations of the central tank (*CT*), the left-wing tank (*LWT*), the right-wing tank (*RWT*), and the surge tank (*SP*) under the manual refuel operation mode. All tanks provide both fuel output via pumps and fuel inlet via valves, each being independently switchable by an external controller. This controller monitors the fuel flow between tanks, calculates the required tank-to-tank fuel transfers and sends the appropriate control signals to the pumps and valves of all tanks.

The internal compositional structure of the MR component is typical for the ground fuel management system. The other operations have a similar compositional architecture; for brevity, Fig. 5.21 contains the details of MR only. Moreover, the component-based implementation of MR exercises most features available in the X-MAN framework, making it a suitable exemplar to illustrate the proposed component-based design and verification approach.

In order to clearly describe the properties under verification, we first clarify the differences between two terms used in the system requirement specifications of our case study. (1) *State*: describes the (durable) condition after the system performs one or a sequence of operations following the entry condition. For instance, the *Inlet Valve* state of each tank shall be either *"SHUT"* or *"OPEN"*. (2) *Status*: It describes the extant condition of some physical component. In this case study, the status of each component is evaluated every cycle and given the value *NORMAL* or *FAILED*.

For verification purposes, most properties that we currently check are *safety* properties, which state that *"something bad must never happen"*, such as:

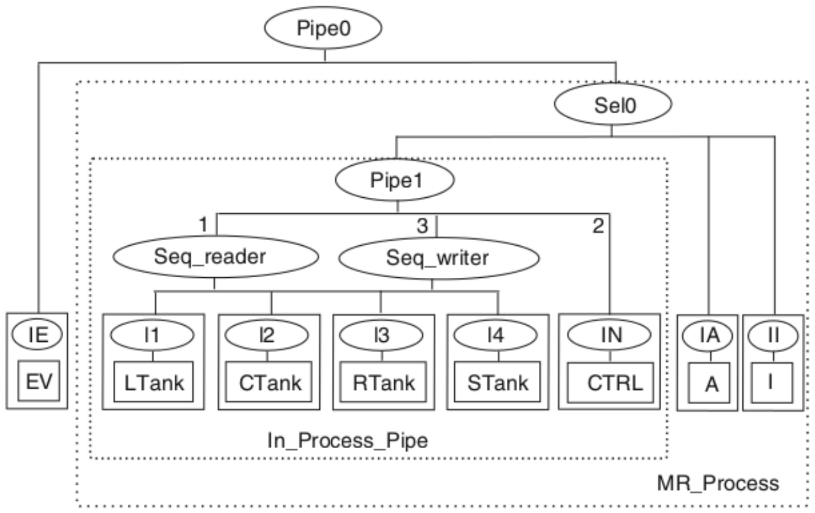


**Fig. 5.22** A component-based manual refuel system

1. When the system *input overflow_condition* is *True*, *MR* must be in the *Abort* operation mode no matter what other inputs are.
2. When the *Inlet Valve* state of a tank is *SHUT*, but the status of this *Valve* is *FAILED*, the fuel mass of this tank must never exceed a certain constant value C.
3. When the status of *MR* is *NORMAL*, the execution of *MR* must never be engaged for longer than 55 s.

These example properties above are at the different abstraction levels. (i) and (*iii*) refer to the entire system-level properties of *MR*, while (*ii*) relates to the specific atomic operation of a tank which can be used directly to formulate the contract of an atomic component.

We implement a component-based version of the ground fuel management system by applying the X-MAN component framework, and verify the given properties of the system by using the compositional verification approach adapted to this framework. The details of the component implementation and compositional verification on the MR operation of this system are presented in the following paragraphs.

### 5.4.2 X-MAN Modeling of Manual Refuel

In X-MAN component modelling, systems are built by composing component instances which are instantiated from component designs in the repository. We will make use of five atomic component designs to model the component-based *MR* system in X-MAN (as shown in Fig. 5.22). These design components are *Tank*, *Controller*, *Evaluator*, and *PrAbort*, and *PrIdling*. *PrAbort* models the computation when an operation mode of the ground fuel transfer system aborts. Similarly, *PrIdling* encapsulates the computation when an operation becomes idle. The final two components are relatively simple in our case study, basically returning the output of the component with some constant values, so, we omit their description here, and concentrate on the remaining three components.

- Tank: This component maintains some stored state variables which represent a mass of fuel in a particular tank. Fuel output and inlet are modelled by the setting of arguments corresponding to each of 3 ports. A positive mass represents fuel input to the tank, negative signifies outputs. The tank component contains an internally selected maximum-mass figure. Input of a fuel mass which causes this figure to be exceeded results in an overflow mass being outputted. The component also outputs the current mass, states and status of pumps and valves of each tank.
- Controller: calculates the fuel flow between four instances of the Tank component, modelling the functionality of the pumps, valves and pipes linking them. The outputs of each tank along with the command signals for each pump and valve are fed to this component.
- Evaluator: accepts the system inputs and evaluates the right operation mode based on the inputs. The evaluation logic is represented as the first-order logic formula in this case. It outputs the evaluation result.

X-MAN model system shown in Fig. 5.22 makes use of eight atomic component instances, and five composite components, involving three different kinds of composition connectors – *Pipe*, *Selector* and *Sequencer*. Firstly, we compose four instances of the design component *Tank*: *LTank*, *CTank*, *RTank*, *STank*, which model *LWT*, *CWT*, *RWT* and *SP* respectively, into a composite component named *TanksReader* using the *Sequencer* connector *Seq_reader*. In the same way, these four component instances are composed together into another composite component named as *TanksWriter* using the connector *Seq_writer*. Thus, every *tank* instance is accessed twice for each iteration of the model. Next, *TanksReader*, an instance of *Controller* named as *CTRL*, and *TanksWriter* are further composed together into a single larger composite component *In_Process_Pipe* by a *Pipe* connector *Pipe1*, which models the *In_Process* operation. Then, *In_Process_Pipe* is composed with the component *A*, which is the instantiation of the design *PrAbort*, and the component *I*, which is the intantiation of *PrIdle*, by the *Selector* connector *Sel0* to construct the more complex composite component *MR_Process*. Finally, by means of another *Pipe* connector *Pipe0*, *MR_Process* is composed with the component *EV*, which is an instantiation of the design component *Evaluator*.

The model behaves as follows at run-time: when a call (with inputs) reaches the top-level connector *Pipe0*, the connector first calls the atomic component *EV* with its required inputs and passes the outputs to the *Sel0* connector. Based on the values passed, *Sel0* evaluates the selection condition in order to choose among the composite components *In_Process_Pipe*, *A* or *I*. If *In_Process_Pipe* is chosen, *TanksReader* is first called and its monitoring outputs are passed to *CTRL*, then finally the computed command ouputs of *CTRL* are fed into *TanksWriter*. When either *TanksReader* or *TanksWriter* is executed, the *LTank*, *CTank*, *RTank*, *STank* are accessed in sequence, but the usage of the outputs from these *Tank* instances is different. For instance, *TanksReader* manipulates the outputs which represent the current mass, the state and status of the tank, while *TanksWriter* uses those outputs that correspond to the handlers of the pump and valve of each tank. If *A* is chosen, the model starts running as the *Abort* operation mode in *MR*. Otherwise, *I* is chosen to simulate the *Idle* behaviour of *MR*.

### 5.4.3 Component-Based Verification of Manual Refuel

Consider the atomic component *LTank* of the *MR* modelling system. It contains six contracts: three of these specify the integer ranges of the outputs parameters, the other two describe the conditions under which fuel may be added without overflow occurring, and the final contract relates to the minimum and maximum execution time of the component. We use one contract *cMassRange* for the component *LTank* as an example:

```
Contract(cMassRange) {
    Inputs:          iAddedMass : int
    Ouputs:          oMass : int, oMaxMass : int
    Pre-condition:   true
    Post-condition:  oMass ≥ 0 ∧
                     oMass ≤ oMaxMass
}
```

Two integer outputs are used in this contract. *oMass* reflects the current fuel mass of the tank, and *oMaxMass* represents the internally selected maximum mass figure. The *Pre-condition* of this contract is *true* which means no assumption of the component inputs is made. The corresponding *Post-condition* requires that *oMass* must always be a positive value and *oMass* must never exceed the value of *oMaxMass* after the execution of the *LTank* component. As a further example, the contract for specifying the extra-functional timing property of this component uses the special parameters *preTime* and *postTime*, which are globally accessible throughout the component, to describe the timing constraint as follows.

```
Contract(cTankTiming) {
    Inputs:
    Ouputs:
    Pre-condition:   true
    Post-condition:  postTime ≤ preTime + 5 ∧
                     postTime ≥ preTime + 1
}
```

According to this contract, the execution time (in seconds) of this contract must always fall within the range $[1, 5]$ under any combinations of inputs. In addition to the basic *Range-Contract* of output variables and *timing-Contract* as introduced above, some component contracts formulate the complex data constraints between model inputs and outputs that are derived from the system requirement. For instance, the contract for the *Controller* component instance *CRTL* is as follows:

```
Contract(cFuelConservationController) {
    Inputs:         iMass[4] : int,
    Ouputs:         oAddedMass[4] : int
    Pre-condition:  iMass[0] ≥ 0 ∧ iMass[1] ≥ 0 ∧
                    iMass[2] ≥ 0 ∧ iMass[3] ≥ 0
    Post-condition: oAddedMass[0] +
                    oAddedMass[1] +
                    oAddedMass[2] +
                    oAddedMass[3] = 0
}
```

In this contract for *CTRL*, the input parameter *iMass* is an Integer array which is passed from the composite component *Seq_reader*. Each element corresponds to the output parameter *oMass* of *LTank*, *CTank*,*RTank*,*STank* respectively. The output parameter *oAddedMass* is also an Integer array which is passed to the composite component *Seq_writer*. And each element corresponds to the input parameter *iAddedMass* of four *Tank* component instances. This *Post-condition* of this contract requires that the fuel mass must be conserved in the *CTRL*.

#### 5.4.3.1 Vertical Verification Phase

We apply the software Model Checker CBMC to formally verify that the implementation of every atomic component in *MR* (a total of eight components) satisfies its corresponding contracts. For instance, we check that the C implementation of the component *CTRL* satisfies the post-condition specified in the contract *cFuelConservationController* given above, under the assumption that the input parameters adhere to the *Pre-condition* constraint. Our verification tool first automatically instruments the following assumption statement at the beginning of the implementation code.

$$\textbf{assume}(cFuelConservation.Pre-condition)$$

and, the following assertion at the end of the code.

$$\textbf{assert}(cFuelConservation.Post-condition)$$

Then, CBMC is applied to verify the instrumented code.

Due to the exhaustive search within the unwinding depth and precise modeling of data variables, CBMC can detect non-trivial corner case bugs, which eluded discovery during testing by means of a conventional test-suite derived from the requirements. As an example, consider the following code fragment from the *CTRL* implementation where the variables have been renamed.

```
const int sink_num = 2; ==****
:::
int val_a = a=sink_num; *
int val_a_remainder = a%b;
:::
if.v_open/{
   val_c = val_a;
}
else{
   val_c = 0;
}
:::
if.val_c > 0/{
   val_c = val_c + val_a_remainder;
}
:::
```

The variable a models a total amount of flow that is distributed between a given number of sinks. The amount is given as an integer quantity, and there can therefore be a remainder (val_a_remainder), which is to be apportioned to the valve flow val_c. Consider the special execution trace when

$a = 1;\ v\_open = 1;$

Then,

$val\_a = 0;\ val\_a\_remainder = 1;\ val\_c = 0;$

i.e., the remainder is non-zero but is not apportioned to the valve flow val_c. The buggy value of val_c propagates to the outputs and causes the violation of the *Post-condition* of *cFuelConservation*. The variables a and v_open are both internal variables. It is difficult for a conventional testing technique to find the primary inputs stimuli that could observe this corner case scenario where a is set to 1 and v_open is set to be *True* at the same time. Consequently, the error was missed by the conventional test suite, while CBMC quickly identifies an erroneous execution trace as above. Analyses revealed that the bug occurred because the code was adopted from a similar algorithm using floating-point arithmetic, and did not handle integer division truncation properly.

### 5.4.3.2 Horizontal Verification Phase

After the contracts of all components have been verified, we can start horizontal verification, which checks the properties of the modeling system based on the verified contracts of the components and their composite relationships. For instance, given a system C composed of two verified components A and B connected by a Selector, we combine the post-conditions in the contracts of A and B as follows:

$$f\ D\ .selC \square\ cond \wedge post_{A}/\ \_\ .:\ selC \square\ cond \wedge post_{B}/$$

where $post_A$ and $post_B$ denote the post-conditions of components A and B, respectively. *selC-cond* is the condition that the sub-component is chosen in the case of using Selector connector. In this example, if *selC-cond* is true, component A is selected for execution; otherwise, component B is selected. For the composite systems using other connectors, the X-MAN-Verifier can also automatically derive the formula f from the proved contracts of sub-components, according to the composition behaviours of the corresponding connectors with respect to the properties under verification.

Then, the X-MAN-Verifier checks whether f implies $post_C$. If so, we have proven that the component *C* satisfies its contract; otherwise the contract may or may not hold. The X-MAN-Verifier can produce a trace that demonstrates how the component C fails to respect its contract. For a given timing property

$$postTime < preTime \text{ C } 55$$

for *MR*, the X-MAN-Verifier returns *hold*. On the other hand, the timing property

$$postTime < preTime \text{ C } 50$$

may be violated and the tool provides a counterexample. We can conclude that the *MR* system is guaranteed to finish execution within 55 time steps, but may exceed 50 time steps. The counterexample extracted is as follows:

```
Inputs :
  in0 = 0; : : : ; in3 = 0;
  preTime = 0;
________________________________
CallingEC
Inputs :
  preTime = 0;
Outputs :
  value = 1;
  postTime = 5;
________________________________
CallingLA
: : : :
________________________________
Outputs :
  out0 = 1; : : : ; out4 = 1;
  postTime = 53;
```

This trace shows the system inputs and outputs, *preTime* and *postTime* of *MR*, and the components on the execution trace with their inputs, outputs, *preTime* and *postTime*.

## 5.5 CMM and the Doors Management System

### 5.5.1 CMM

The concepts of the CMM are related to the Heterogeneous Rich Components (HRC) which were developed in the SPEEDS project [21, 38, 52, 96]. In CESAR, the contract-based specification methodology and data model for component based development has been adopted and refined to support the CESAR development process. Contracts refer not to the internal states or the internal behaviour of a component but to the observable interactions via its interfaces. The interfaces of a component are specified by a set of (observable) variables. Those variables, called flows, can be characterized as input, output or bidirectional from the components point of view. In addition also a data type can be assigned for each flow. Contracts argue about traces of this interface variables.

In the first place a component is a black box with a sound specified interface and without any dynamic aspects like behaviour or interaction protocol (no traces). Since the interaction is the observable part of the behaviour of the component it is specified in the same manner as the (non-) functional behaviour of the component itself. Behaviour is specified by mechanisms which decide if a certain trace ("run") is part of the behaviour or not. A run is a possibly infinite trace of variables assigned to the variables. Finite state machines are one possible mechanism to determine if such a trace is part of the behaviour or not. This generic approach allows to specify the behaviour of a very wide range of components. Such components could be software or hardware elements because the specification takes only the observable variables (software variables or hardware pins) at the interface of the component into account. The concept of a contract for the assumed behaviour of the environment of the component and the promised behaviour of the component allows to verify if a implementation fulfils the specification. Note that this implies that the execution framework of the component is in most cases essential to provide the promised behaviour. For example, a software function does not automatically calculates $a + b$ but does this only if it is called (e.g. "$c = f(a, b)$").

A component is a black box with a generic and formal specification of behaviour based on traces of observable variables. Control events, like triggers, are modelled explicitly and also part of the contracts of a component. The assumption-promise pairs allow to verify the implementation of the component independent from a certain environment. Furthermore, this enables reuse of component implementations. The formal nature of the specification of the components allows a so called Virtual Integration Test (VIT) without the need to implement any of the involved components, see [40]. To support the creation of this formal specification a pattern-based specification language has been developed in SPEEDS as well as in CESAR. This language supports the user to formalize requirements, see requirement formalization in Sect. 3.3.

Composition of CMM components is the parallel execution of the subcomponents. What does that mean? First of all this is not true because the parallel operator is not the only composition operator defined in the semantic definition but it is the most important one. Second, not the components are executed but the sets of allowed traces are merged to result in the composed set. Since the formal specification is driven by the idea to have state machines which define these sets of traces the parallel composition of this state machines results in one state machine which defines the set of traces of the composition. Due to this approach, the connections between the subcomponents define which variables match between two or more component interfaces. This is necessary because the alphabets of the traces have only local scope. Note that this implies identity of connected variables. If data or event exchange has a certain behaviour this has to be represented by a additional component, e.g. a bus or queue.

To summarize, the specification is contract-based and supports reuse of a wide range of (hardware and software) components. The formalization of requirements enables early assessments of requirements as presented in Sect. 3.4.

The result is the definition of the CMM as common views to data through a heterogeneous tool and language environment, the RTP (see Sect. 6). Based on this common view to different development entities (software components, requirements, test cases, ...), a process library was developed to enable the users to instantiate a company specific development process for safety-critical embedded systems. A generic development process pattern containing four generic development phases was identified to be applicable in several domains and companies.

The following subsection relates the application of the CMM methodology and data model to the Doors Management System.[6] This example covers two of these phases to illustrate the CESAR methodology.

### 5.5.2 Application to the DMS

The Doors Management System example is used to show how this process could be instantiated. This system is highly safety-critical because significant aircraft accidents and incidents occurred due to uncontrolled decompression. In some of them doors not fully closed, latched and locked opened in flight, leading to an explosive decompression. The instantiated example development process contains four analysis and design phases, where all referenced development entities are described in term of the CMM.

The top level requirements of the DMS are listed below.

1. The DMS shall be able to provide to pilots the status of Aircraft Door Slides. (Functional-Perspective)

[6] Odile Laurent and François Pouzolz; Airbus France; Airbus generic pilot application Aircraft doors management system; CESAR Use Case; May 2009

2. The DMS shall inhibit pressurization of Aircraft Doors when Doors are not fully closed and locked. (Operational Perspective)
3. The DMS shall enable pressurization of Aircraft Doors when Doors are fully closed and locked. (Operational Perspective)
4. When in Flight, the DMS shall inhibit intentional or unintentional opening of aircraft doors. (Operational Perspective)
5. When on ground, the DMS shall allow opening of Aircraft Doors. (Operational Perspective)

The whole set of top level requirements addresses certain concerns of the DMS system. We focus on a subset only to simplify the example. The top level requirements are transformed to CMM *SystemRequirements* by specifying

- What are the assumptions to the environment?
- What is the promised behaviour which the component can guarantee?
- To which Perspective is the requirement associated?
- To which Aspect is the requirement associated?
- Which component(s) shall satisfy the requirement?

In this early phase, having only a very abstract representation of the DMS, all requirements are promises since the environment is not clearly defined. Also, the DMS itself is the components which shall satisfy all the requirements. The definition of the system boundaries, the refinement of the SystemRequirements and the decomposition of the system follow during the development process. The association of the requirements to Perspectives and Aspects is indicated in brackets after the textual representation of the requirements.

#### 5.5.2.1 Operational Analysis

In this first phase of the development process the top level requirements are represented as SystemRequirements and are connected via *Satisfy* links to one *OperationalActor* the DMS representation in the operational Perspective.

The Operational Capability Specification is performed with the goal of deriving operational scenarios and functional requirements that cover the original requirements. Since in this step the DMS is sketched in its boundaries, realistically many functional and non-functional SystemRequirements will have to be reconsidered in later steps.

First, the OperationalActors are assessed from the SystemRequirements. In addition to the DMS the other OperationalActors Passenger, Pilot, Cabin Doors, Crew and Landing Gear System are identified.

As a second step, operational capabilities were derived from the SystemRequirements by a system engineer such as "Enable Pressurization when Doors locked" or "Keep door locked when in flight" (see Fig. 5.23). For completeness and traceability these SystemRequirements are related explicitly to the operational capabilities by Satisfy links.

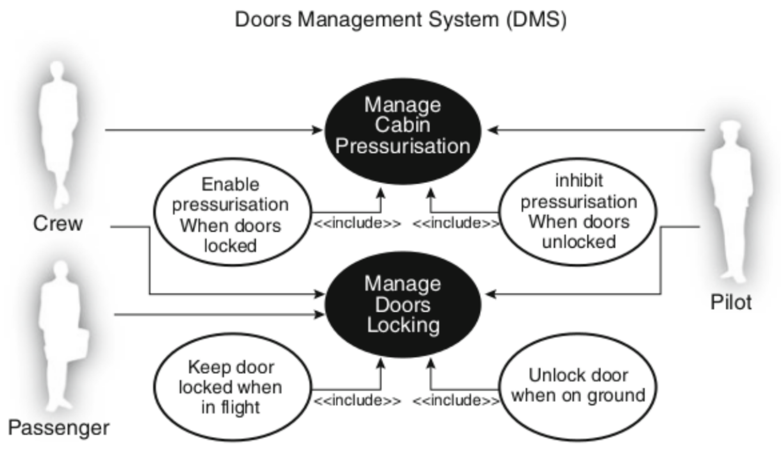

Doors Management System (DMS)
Crew
Passenger
Pilot
Manage Cabin Pressurisation
Manage Doors Locking
Enable pressurisation When doors locked
inhibit pressurisation When doors unlocked
Keep door locked when in flight
Unlock door when on ground
<<include>>
<<include>>
<<include>>
<<include>>

**Fig. 5.23** Use case diagram of the operational capabilities of the DMS

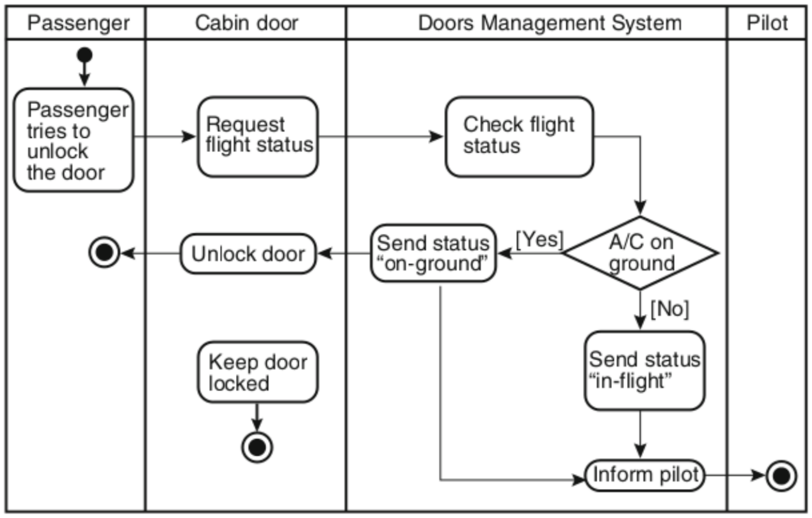


**Fig. 5.24** Operational scenario of the DMS

The use-case-driven scenario description is refined by several scenarios which has to be supported. One of this scenarios is presented in Fig. 5.24.

Based on the identified activities, operational processes are specified which orchestrate these activities. For the operational process, the interaction between the operational activities is defined. The description of such uses cases allows sketching the DMS in its system boundaries. That means that the interaction with other actors such as passengers, pilot, crew and cabin door is identified. New SystemRequirements are derived form the decomposition of the operational specification into several activities. These new SystemRequirements (usually) refer to the functional Perspective.

In this design step also a preliminary hazard analysis is performed to identify the safety-relevant operational activities. You may find further information about this in Sect. 2.5.

#### 5.5.2.2 Functional/Non-functional Analysis

Input of this design step is the result of the operational analysis. In particular this consists of operational scenarios, user requirements which could not yet be refined, hazards as well as their classifications and functional SystemRequirement identified during the operational analysis.

The goal of the functional/non-functional need specification is to identify and specify functions and where appropriate refine them by defining sub-functions. The DMS is decomposed into the functions “Manage Cabin Pressurization” and “Manage Doors Locking”. For traceability all the defined functions have a relation (e.g. CMM *Derive* link) to the design entities defined in preceding phases.

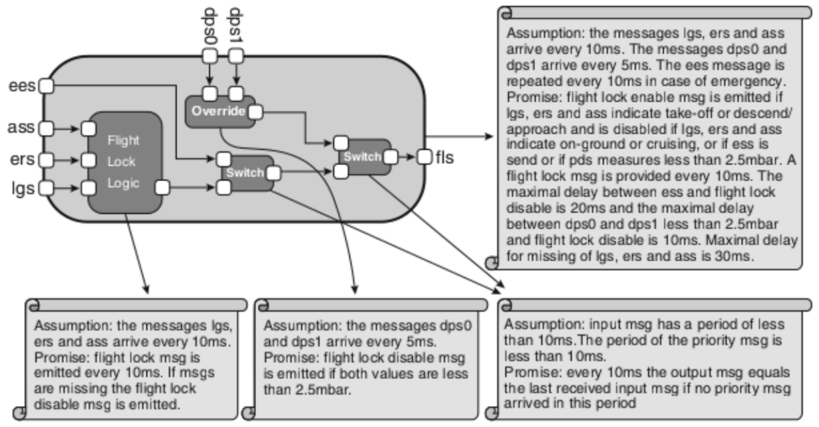
One intermediate step in before mapping the functions to physical elements is to map them to logical elements. This logical components represent a preliminary architecture of the whole system before determining certain implementation solutions. One the one side the components are not characterized as hardware or software units but have interface discretions and allow to refine the expected behaviour in a abstract manner.

We illustrate this taking the “Manage Doors Locking” function as example. This function shall be represented by a logical component named “FlightLockController” which controls the flight lock on each door of the DMS. The purpose of this component is to signal to the flight lock actuators if the lock shall be enabled or disabled. Two of the top level requirements apply to this component and therefore the behaviour of the component shall be to enable the lock during take-off and landing/approaching and to unlock it during cruse and if on ground. This status is provided by the Landing Gears System, the Engine Status and the Air Speed Sensors. In addition three safety-related requirements enforce to disable the lock:

1. If the signals “landing gear status” (lgs), “engine running status” (ers) and “air speed status” (ass) are not available or
2. If the “emergency evacuation” signal (ees) is send or
3. If the differential pressure is less than 2.5 mbar.

The differential pressure is measured by two independent sensors and shall enable emergency evacuation on ground and doors can be opened without risk (low differential pressure) even if the other systems fail. Also some timing constraints are added to these components. The overall requirement for the “FlightLockController” is on the right side of Fig. 5.25.

The “FlightLockController” is decomposed into the “FlightLockLogic” acquiring the flight status and emitting the “flight lock status” (fls) which could be



Assumption: the messages lgs, ers and ass arrive every 10ms. The messages dps0 and dps1 arrive every 5ms. The ees message is repeated every 10ms in case of emergency. Promise: flight lock enable msg is emitted if lgs, ers and ass indicate take-off or descend/approach and is disabled if lgs, ers and ass indicate on-ground or cruising, or if ess is send or if pds measures less than 2.5mbar. A flight lock msg is provided every 10ms. The maximal delay between ess and flight lock disable is 20ms and the maximal delay between dps0 and dps1 less than 2.5mbar and flight lock disable is 10ms. Maximal delay for missing of lgs, ers and ass is 30ms.

Assumption: the messages lgs, ers and ass arrive every 10ms. Promise: flight lock msg is emitted every 10ms. If msgs are missing the flight lock disable msg is emitted.

Assumption: the messages dps0 and dps1 arrive every 5ms. Promise: flight lock disable msg is emitted if both values are less than 2.5mbar.

Assumption: input msg has a period of less than 10ms.The period of the priority msg is less than 10ms. Promise: every 10ms the output msg equals the last received input msg if no priority msg arrived in this period

**Fig. 5.25** Logical view to the "FlightLockController" and its requirement

"enabled" or "disabled", the "Override" component which sends the fls signal with content "disabled" if the differential pressure is low and two switches which pass through the message available at the "normal" port if no message is available on the priority port in a certain time frame. The priority message is passed if available and the other message is ignored. Note that no message queues are assumed and therefore only one message per port is recognized. For implementation, these messages and ports could be realized by variables with certain values as well as pins with high or low current. This abstract view of the "FlightLockController" decomposition is presented and also the requirements of the parts of the components are shown. All the requirements are formulated as SystemRequirement (with Assumption and Promise) to differentiate between what the components are responsible for and what they assume/need from the environment. The interfaces of the components describe what kind of exchange is possible, e.g., if boolean values (boolean variable, high/low current) or numerical values (integer, 8-bit coded data).

Note that this is specification and not implementation. The components mentioned can not be compiled into code or hardware but only refer to a implementation. Therefore not the components itself but the referenced implementation has to fulfil the requirements of the component. The SystemRequirements of these components argue about traces for the interface elements. If an implementation produces the same output trace as specified it fulfills the SystemRequirement. This could only be achieved if the input trace is also conform to the assumed one, since the implementation reacts to the inputs.

Note that in Sect. 3.4 a more detailed description of how a natural language requirement can be formalized into a formal (text-)pattern-based SystemRequirement. Such SystemRequirements describe sets of traces for Assumption and Promise pairs in a formal but human readable way.

In this example, we assume that such a formalization has been made and for each component the traces are defined e.g. using finite state machines (FSM). For each

SystemRequirement one set of traces has been defined for the Assumption and one other for the Promise. These traces argue about the values of the interface variables and do not take internal states into account. For the three different types of (sub-) components of the "FlightLockController", we assume one SystemRequirement per component and one SystemRequirement for the "FlightLockController" itself. In this example none of the sub-components is connected to all of the interfaces of the "FlightLockController" component and the "switch" is reused.

The question we want to address is: Does this composition of the four sub-components (three different types) fulfill the SystemRequirement of the "Flight-LockController"? The answer is of course yes, but why?

The traces of the "FlightLockController" can be represented using FSMs. The same could be done for the three different types of sub-components. The problem is how to determine that the different FSMs are equivalent to the "FlightLockController" ones. Each FSM has its own alphabet since this relates to the interface of the component only. Connections between instances of this component types create a mapping relation between the different alphabets. In addition for each instance of the same type, an individual FSM is used. Build up on this the parallel composition of the FSMs can be created and this composed FSM defines the set of traces of the composition. At this point the FSM of the "FlightLockController" and of the composition can be compared if the composition contains at least the same traces as the "FlightLockController" FSM. We call this relation "entailment" and via the CMM the SystemRequirements and components are linked with each other to enable traceability of this relation. Note that because of merging the traces the connections between the components can not have certain properties.

The purpose of the entailment relation is not only to check if the composition is valid but to reduce the integration testing effort. If entailment is given, the individual sub-components have to be checked and not the composition itself. This also reduces the effort for verification of reusable elements since their implementation must only be checked once for each type and not for each instance in a certain composition as well. In addition this allows e.g. an OEM to contract several suppliers and to ensure that the integration is valid without exhaustive testing against each of the implementations.

This relation does not only support this design step but is applicable in all design steps of the CESAR design process. In Sect. 3.4, as mentioned before, guidelines as well as tools support this kind of (de-)composition also on requirements wrt. only a very draft architecture. The overall goal of this methodology is not only to reduce verification effort but also to enable traceability from top level requirements down to the implementations of individual components as well as the overall composed system. One last important aspect is the analysis of the impact of a change of requirements in all design steps (refer to Sect. 3.4 for details).