Analytical software design: introduction and industrial experience report
Osaiweran, A.A.H.; Boosten, M.; Mousavi, M.

Published: 01/01/2010

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
- A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 06. Dec. 2018
Analytical Software Design: Introduction and Industrial Experience Report

Ammar Osaiweran$^{1,2,*}$, Marcel Boosten$^2$, MohammadReza Mousavi$^1$

1 Eindhoven University of Technology, Eindhoven, The Netherlands
2 Philips Healthcare, CardioVascular X-Ray, Best, The Netherlands

a.a.h.osaiweran@tue.nl, marcel.boosten@philips.com, m.r.mousavi@tue.nl

Abstract

Analytical Software Design (ASD) is a design approach that combines formal and empirical methods for developing mathematically verified software systems. Unlike conventional design methods, the design phase is extended with more formal techniques, so that flaws are detected earlier, thereby reducing the time needed for coding, testing, and integration. In this paper, we demonstrate formal details and concepts behind the ASD approach, report about our experience with applying ASD in industrial control applications within Philips Healthcare, and discuss results and findings gathered during this work as well as some commonly faced issues and their practical solutions.

1 Introduction

Software projects are too often subject to failure and part of the reproach goes to the lack of established rigorous and predictable methods for software development. Even selection of an appropriate design method and tool is a challenging task, simply because there is a variety of different approaches with their own practices, processes, notations and tools. Designing software still depends on human creativity and experience, and this often results in ad-hoc and clever solutions to technical problems. As the time progresses, clever solutions bring challenges for testing, integration, and quality measurements. Extensibility, maintainability, and reliability are debased and compromised. Formal methods can come to the rescue to facilitate applying structured, rigorous and provably correct solutions, which enhance the above-mentioned measures.

Analytical Software Design (ASD) is a design approach that combines the application of formal mathematical methods such as Sequence-Based Specification (SBS) [8, 6, 13], Communicating Sequential Processes (CSP) [5, 16, 3] and the model checker Failure Divergence Refinement (FDR2) [3] with software development methods such as the Box Structure Development Method (BSDM) [10, 15], Stepwise Refinement, and Component-Based Software Development. The ASD approach is currently applied in a wide spectrum of industrial applications [1, 2].

Practically, ASD targets complex control software and, in general, event-driven systems where concurrency plays a crucial role. Such systems are known to pose serious challenges for conventional design methods. ASD can be applied for verifying protocols among layers in a given

$^*$Sponsored by Philips Healthcare, Best, the Netherlands.
$^1$Supplied by Verum Consultants BV, the Netherlands.
architecture by considering the interface-level behavior and abstracting from implementation, e.g.,
non-functional and data-related, details.

In this report, we introduce the ASD concepts formally and informally, demonstrate our gathered experience with applying ASD in Philips Healthcare, and we discuss some suggestions to extend ASD features.

To illustrate the concepts of ASD, we apply the technology on the design of a case-study inspired from our industrial application domain, described below.

**Case Study**  Our case study concerns a typical trolley system, on which a patient can lie. The trolley can be attached or detached to the body of a system: an MRI device, for instance. Figure 1a. depicts the trolley and its parts. The system includes the following components:

1. A force sensor: an On/Off sensor. A signal is sent when the trolley is docked to the body of the system.
2. A lock: an On/Off lock to attach/detach the trolley to the body. The Lock must be applied when a trolley is docked to the body.
3. A motor: a bi-directional motor with built-in brakes. The mutual safety between the breaks and the motor is assumed built-in. The motor moves the trolley platform to the right or to the left.
4. A position sensor: two sensors at the right-most and the left-most positions inside the body to detect the position of the platform of a docked trolley.

![Figure 1: A trolley and a user console](image)

The following list exhibits the requirements on the trolley:

1. The controller is responsible of initializing and uninitializing software and hardware components. The controller ensures that components are successfully (or failed to be) initialized.
2. When the trolley is docked to the body, the lock is applied automatically.

3. The platform of the trolley can move to the left or to the right via the Left and the Right buttons, respectively.

4. The movement is not allowed, if the trolley is not attached to the body. This is to prevent patients from tumbling over.

5. When the Undock button is pressed, the trolley is uncoupled from the body.

6. The trolley can be uncoupled only if it is at the leftmost position and the Undock button is pressed.

The goal of this case-study is to demonstrate different aspects of the ASD technology by applying it to the design and verification of the trolley specified above.

Structure of The Report  The rest of this report is organized as follows. Section 2 details the required preliminary concepts, and demonstrates methods, languages and tools used throughout this report. In Section 3 an overview of ASD models is given. Section 4 details the ASD specification; models of run-time interaction and communication among ASD components are further detailed in this section. In Section 5 we briefly show how CSP models are combined into a model for verification, we explain the importance of lazy and eager abstractions for interface verification, and we show how FDR can be used for incremental development of software systems. In Section 6 we detail our observations about the usage of ASD in our domain. In Section 7 we discuss the steps taken to develop our case study. In Section 8 we discuss some common modeling issues when modeling components in ASD, and provide some suggestions to circumvent them. Finally, we conclude our paper in Section 9.

2  Background

A software design developed in ASD comprises components (developed top-down, bottom-up, or middle-out) that interact with one another or their environment via channels. System functionality is distributed and decomposed among components, and a majority of these components are distributed in a hierarchical structure to divide and manage the development and verification effort of these components separately [11].

An ASD specification, which comprises SBS, is used to construct models of functional behavior of ASD components. From ASD models, formal models (CSP) and source code implementation (C++ or C#) can be generated automatically. Via this mechanized transformation, ASD guarantees consistency between specification, verified models, and implementation code of ASD components (this is called the ASD triangle). ASD specifications are coupled with an industrial-strength tool, called the ASD ModelBuilder. In ASD, the external behavior of a component or system (at interface-level) is represented by an ASD interface model. This model is employed as an abstraction for verification via model checking to combat the state space explosion problem and used as a building block for refinements of subsystems. Concrete ASD components are specified via another model, called the ASD design model.

Via model checking, a concrete ASD component can be verified for the absence of deadlocks, livelocks and illegal calls with only the interface models of its used components and without considering the details of their refined concrete components. Verification is done component-wise until the system is developed.
2.1 Interfaces and the stepwise refinement technique

Refinement can be seen as a transformation of a high-level abstract specification of a system to one or more detailed subsystems. The stepwise refinement technique aims at refining a system specification in stages until its implementation is developed. Each stage entails breaking a system down into subsystems which only need to be verified against the specification constructed in the previous step. In practice, refinements allow one to split a system into subsystems handled by independent designers.

By means of refinements, designers can develop a system in a top-down fashion, where a high-level abstract specification of the system is first created. This specification often describes how the system can be used and interact with its environment. All possible inputs are addressed with possibly lower-level components as stubs. The specification is then refined into subsystems which in turn are refined to address further details. As a consequence, the system can be depicted as a structure of block-diagrams with interconnections that define their communication graph. This ultimately represents the system as a hierarchy where leafs depict the base concrete implementation of the system, see Figure 2.

![Figure 2: A system and its hierarchical structure](image)

On the other hand, via the bottom-up approach one can start developing a system from implementation. This can be established by composing existing reusable system components such as legacy code or third party components. In software development processes, designers tend to combine the top-down and bottom-up approaches. In case existing components are reused in the development of software, components can be represented by the behavior of their interfaces.

Constructing an interface specification from a detailed component or system requires a special kind of abstraction which is dual to refinement. Extracting such an abstract specification enables compositional verification through testing or model checking.

Both refinement and abstraction techniques allow deriving a software system by formal and systematic steps, and allow compositional design and verification processes (a specification can be used once to design a system and once another to represent the system in verification with others).

Notice that $S$ in Figure 2 comprises concrete components that form a network based on their communication graph. Furthermore, $S$ can be represented as a tree of subsystems where leafs reveal its concrete components. For instance, concrete components of $Q$ are distributed as a hierarchy while concrete components of $S$ are not. System $S$ might be represented by a single interface specification with respect to its concurrent input ports behavior or an abstract specification which might include also the output port behavior. This clarifies the widely held misconception that stepwise refinement is only appropriate to develop systems with hierarchical structures.

**Case Study:** consider our case study introduced in Section 1. In order to manage the design of the controller for the trolley, we refine it into the hierarchy depicted in Figure 3. The top-
The most component is the system controller (SC), which interacts with the user console, while the bottom-most components are logical devices representing interaction with the controllers of their physical counterparts, such as the sensors, the lock and the motor. These devices are controlled independently by both the Docking Controller (DC) and the Movement Controller (MV).

**Figure 3: Component Architecture of The Trolley Controller**

### 2.2 Sequence-Based Specification Method

In this section we provide a brief and global overview of the Sequence-Based Specification (SBS) method, and we base ourselves on [15, 7] in what follows in this section. The reader who is not interested in SBS details can safely skip this section and proceed to Section 2.3. In Section 4 the description of SBS is refined to the extent related and used in ASD.

The Sequence-Based Specification method is used as a vehicle for specifying behavior of systems developed using the box-structure development method (BSDM). SBS provides a formal approach for specifying functional behavior of components with total, complete specification traceable to informal requirements. Such requirements are numbered and referenced in SBS tables forming the concept of requirements’ traceability (SBS forces a tabular form for specification). The following template is used to tag requirements for later references during the specification process.

<table>
<thead>
<tr>
<th>Tag No.</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC,R1</td>
<td>The controller is responsible to initialize and uninitialize software and hardware components. It ensures that components are successfully (or failed to be) initialized.</td>
</tr>
<tr>
<td>DC,R2</td>
<td>When the trolley is docked to the body, the lock is applied automatically.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 4: Tagged requirements**

The main benefit of SBS as a formal specification method is that designers follow the same design and specification steps, so that a component behavior becomes easily understood among designers who understand the method. This is unlike conventional design methods where software is informally documented and specified, so that critical design details might be missing in case newly introduced designers are not involved in certain design decisions. It is expected that more time and effort are spent for domain and knowledge transfer in conventional approaches compared to those of SBS.

To specify component behavior using SBS the component boundary must be clearly identified. Component behavior is constructed in terms of actions. Stimuli and responses are control and information flow that cross the boundary of the component. A system or a component behavior can be specified in three steps: black-box, state-box, and clear-box.
The Black-Box  A black-box specifies the external behavior of a component. A black-box is a function \( BB : S^* \rightarrow R \) that maps a stimuli sequence to a single response, where \( S^* \) denotes the set of all finite stimuli sequences and \( R \) is the set of all responses. In a black-box specification, all possible input stimuli sequences are enumerated in the order of their length starting from the empty sequence. The black-box specification is organized in a table (nested tables if a design is complex), where each row represents a stimuli sequence with a proper response. Below we consider the Docking Controller component of our case study to give an example of the sequence enumeration for the black-box specification.

Case Study: in the following, we detail the black-box sequence enumeration of the Docking Controller. The stimuli and responses used in the black-box sequence enumeration process are listed in Figure 5.

\[\text{Stimuli}(S)\]

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Interface (channel)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>IDockingController</td>
<td>The Initialize request sent from the System Controller via the IDockingController interface.</td>
</tr>
<tr>
<td>UnInitialize</td>
<td>IDockingController</td>
<td>The UnInitialize request.</td>
</tr>
<tr>
<td>ReleaseLock</td>
<td>IDockingController</td>
<td>A request to release the lock.</td>
</tr>
<tr>
<td>DockingConfirmed</td>
<td>IDockingController</td>
<td>SC confirms receiving the docking status callback sent by DC.</td>
</tr>
<tr>
<td>isDockedNow</td>
<td>ISensorCB</td>
<td>When a platform is attached, the Sensor (driver) informs DC the status via this callback.</td>
</tr>
<tr>
<td>isDocked</td>
<td>ISensorCB</td>
<td>The Sensor replays DC query that the platform is docked.</td>
</tr>
<tr>
<td>isUnDocked</td>
<td>ISensorCB</td>
<td>The Sensor replays DC query that the platform is not docked.</td>
</tr>
<tr>
<td>UnInitialized</td>
<td>ISensorCB</td>
<td>The Sensor is successfully uninitialized.</td>
</tr>
</tbody>
</table>

Figure 5: The set of stimuli of the Docking Controller

Figure 6 depicts a List of responses of the Docking Controller component.

\[\text{Responses}(R)\]

<table>
<thead>
<tr>
<th>Response</th>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitializeDevices</td>
<td>ISensor, ILock</td>
<td>An abstract response denotes initializing both the Sensor and the Lock devices.</td>
</tr>
<tr>
<td>UnInitializeDevices</td>
<td>ISensor, ILock</td>
<td>Uninitializing both devices.</td>
</tr>
<tr>
<td>ConfirmSensor</td>
<td>ISensor</td>
<td>Confirms the Sensor of receiving the docking state.</td>
</tr>
<tr>
<td>SysCtrCBDocked</td>
<td>ISystemCtrCB</td>
<td>Informs SC of a docked platform.</td>
</tr>
<tr>
<td>SysCtrCBnotDocked</td>
<td>ISystemCtrCB</td>
<td>The SC is informed that the platform is not docked.</td>
</tr>
<tr>
<td>SysCtrCBUnInitialized</td>
<td>ISystemCtrCB</td>
<td>All devices are successfully uninitialized.</td>
</tr>
<tr>
<td>LockReleaseLock</td>
<td>ILock</td>
<td>Request to release the lock.</td>
</tr>
<tr>
<td>NtfyCtr_AplyLk_CnfrmSnsr</td>
<td>ISystemCtrCB, ILock, ISensor</td>
<td>Abstract response denotes notifying the System Controller that the platform is docked, applying the lock, and confirming the Sensor.</td>
</tr>
<tr>
<td>Illegal</td>
<td>-</td>
<td>Invoking a stimulus is illegal.</td>
</tr>
<tr>
<td>Null</td>
<td>-</td>
<td>No response.</td>
</tr>
</tbody>
</table>

Figure 6: The set of responses of the Docking Controller
Notice that a response can represent a collection of actions. For instance the InitializeDevices response represents initializing both the used sensor and lock components. As an advantage, the SBS method forces designers to think abstractly during the specification process to prevent any potential over-specification. Figure 7 depicts the enumeration process of stimuli sequences that forms a abridged black-box specification. Notice that new derived requirements (denoted by D1 and D2) might be introduced during the enumeration process resulting into an increase in the precision of requirements.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Response</th>
<th>Equivalence</th>
<th>Tag No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty Zero</td>
<td>Null</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>All devices are initially uninitialized.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length One</td>
<td>Initialize</td>
<td>InitializeDevices</td>
<td>DC_R1</td>
</tr>
<tr>
<td></td>
<td>UnInitialize</td>
<td>Illegal</td>
<td>D1</td>
</tr>
<tr>
<td></td>
<td>isDocked</td>
<td>Illegal</td>
<td>D1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length Two</td>
<td>Initialize, isDocked</td>
<td>SysCtrCBDocked</td>
<td>D2</td>
</tr>
<tr>
<td></td>
<td>UnInitialize</td>
<td>Illegal</td>
<td>DC_R2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length Three</td>
<td>Initialize, isDocked</td>
<td>UnInitializeDevices</td>
<td>DC_R1</td>
</tr>
<tr>
<td></td>
<td>UnInitialize</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length Four</td>
<td>Initialize, isDocked</td>
<td>SysCtrCBUnInitialized</td>
<td>DC_R1</td>
</tr>
<tr>
<td></td>
<td>UnInitialize, UnInitialized</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length Five</td>
<td>Initialize, isDocked</td>
<td>InitializeDevices</td>
<td>DC_R1</td>
</tr>
<tr>
<td></td>
<td>UnInitialize, UnInitialized, Initialize</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: Sequence enumeration of DC black-box: abridged specification

The black-box specification excludes any state data between the stimuli sequences. The black-box specification can be reduced by partitioning it into equivalent classes each of which represents a canonical stimuli sequence. Such canonical sequences are formed during the enumeration process of stimuli sequences: two sequences \( s, t \in S^* \) are behaviorally equivalent (Mealy equivalent [9]), denoted by \( s \equiv_{\text{Mealy}} t \), when all possible non empty extensions of both sequences lead to a same response. During enumeration such equivalent sequences are noted in the table. From the table above, an example of two behaviorally equivalent stimuli sequences is the sequence \( \langle \text{Initialize} \rangle \) of length one and the sequence \( \langle \text{Initialize}, \text{isDocked} \rangle \) of length five, since both behaviorally lead to the same response and the same future extension.

Calculating such canonical sequences reduces a lengthy table to a number of smaller tables each of which represent a canonical sequence (which can also be seen as a Mealy state). In [15], the table format depicted in Figure 8 is used to capture the black-box specification for each constructed canonical sequence.
Case Study: in the following, we detail the black-box specification with canonical sequences. Consider the specification of DC depicted in Figure 7. The black-box specification after calculating the canonical sequences through the enumeration process is depicted in Figure 9.

### UnInitialized :<>

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Response</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Initialize_Devices</td>
<td>CheckDockingStatus</td>
</tr>
<tr>
<td>UnInitialize</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>ReleaseLock</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>DockingConfirmed</td>
<td>Null</td>
<td>UnInitialized</td>
</tr>
<tr>
<td>IsDockedNow</td>
<td>ConfirmSensor</td>
<td>UnInitialized</td>
</tr>
<tr>
<td>IsDocked</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>IsUnDocked</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>UnInitialized</td>
<td>Illegal</td>
<td>-</td>
</tr>
</tbody>
</table>

### CheckDockingStatus :< Initialize >

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Response</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>UnInitialize</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>ReleaseLock</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>DockingConfirmed</td>
<td>Null</td>
<td>CheckDockingStatus</td>
</tr>
<tr>
<td>IsDockedNow</td>
<td>ConfirmSensor</td>
<td>CheckDockingStatus</td>
</tr>
<tr>
<td>IsDocked</td>
<td>SysCtrCBDocked</td>
<td>Idle</td>
</tr>
<tr>
<td>IsUnDocked</td>
<td>SysCtrCBnotDocked</td>
<td>Idle</td>
</tr>
<tr>
<td>UnInitialized</td>
<td>Illegal</td>
<td>-</td>
</tr>
</tbody>
</table>

### Idle :< Initialize, isDocked >

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Response</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>UnInitialize</td>
<td>UnInitialize_Devices</td>
<td>-</td>
</tr>
<tr>
<td>ReleaseLock</td>
<td>LockReleaseLock</td>
<td>Idle</td>
</tr>
<tr>
<td>DockingConfirmed</td>
<td>Null</td>
<td>Idle</td>
</tr>
<tr>
<td>IsDockedNow</td>
<td>NtfyCtr_AplyLk_CnfrmSnsr</td>
<td>Idle</td>
</tr>
<tr>
<td>IsDocked</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>IsUnDocked</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>UnInitialized</td>
<td>Illegal</td>
<td>-</td>
</tr>
</tbody>
</table>

### UnInitializing :< Initialize, isDocked, UnInitialize >

<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Response</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>UnInitialize</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>ReleaseLock</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>DockingConfirmed</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>IsDockedNow</td>
<td>Null</td>
<td>UnInitializing</td>
</tr>
<tr>
<td>IsDocked</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>IsUnDocked</td>
<td>Illegal</td>
<td>-</td>
</tr>
<tr>
<td>UnInitialized</td>
<td>SysCtrCBUnInitialized</td>
<td>UnInitialized</td>
</tr>
</tbody>
</table>

Figure 9: The black-box specification of DC with canonical sequences
The State-Box

The state-box specification expands the black-box specification (tools can be used for automatic generation) such that state data are introduced. The state-box comprises transparent state data but is still implementation independent (no source code is written or generated yet). The state-box function is a mapping from a single stimulus and an initial state to a response and a next state, written $SB : (Q \times S) \rightarrow (Q \times R)$. The state-box specification must be behaviorally equivalent to the black-box specification. In the state-box specification using SBS, state variables are introduced through a process called the canonical sequence analysis [15]. Briefly, it means that the values of such variables identify each state in the state-box specification. The state-box is a set of tables each of which is dedicated to a stimulus. Such tables must reflect the same functional behavior specified in the black-box specification.

Case Study: we detail the state-box of the Docking Controller. In Figure 10 we map each canonical sequence to its characteristic predicate and a state name.

<table>
<thead>
<tr>
<th>Canonical sequence</th>
<th>Distinctive predicate</th>
<th>State name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;\rangle \rho_{\text{st}}$</td>
<td>initialized=false ∧ docked=false ∧ ready=false ∧ uninitializing=false</td>
<td>UnInitialized</td>
</tr>
<tr>
<td>$&lt;\text{Initialize}&gt; \rho_{\text{st}}$</td>
<td>initialized=true ∧ docked=false ∧ ready=false ∧ uninitializing=false</td>
<td>CheckingDockingStatus</td>
</tr>
<tr>
<td>$&lt;\text{Initialize}, \text{isDocked}&gt; \rho_{\text{st}}$</td>
<td>initialized=true ∧ docked=true ∧ ready=true ∧ uninitializing=false</td>
<td>Idle</td>
</tr>
<tr>
<td>$&lt;\text{Initialize}, \text{isDocked}, \text{Uninitialize}&gt; \rho_{\text{st}}$</td>
<td>initialized=true ∧ docked=true ∧ ready=true ∧ uninitializing=true</td>
<td>UnInitializing</td>
</tr>
</tbody>
</table>

Figure 10: Calculating characterizing predicates for the canonical sequences

Figure 11 depicts the state-box specification of both the initialize and ReleaseLock stimuli. The rest of the stimuli are not specified here. Notice that, when constructing the state-box specification, both the state-box specification and the black-box specification should exhibit the same behavior.

<table>
<thead>
<tr>
<th>Stimulus : initialize</th>
<th>Current state</th>
<th>Response</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnInitialized</td>
<td>Illegal</td>
<td>Sink State</td>
<td></td>
</tr>
<tr>
<td>CheckDockingStatus</td>
<td>Illegal</td>
<td>Sink State</td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>Illegal</td>
<td>Sink State</td>
<td></td>
</tr>
<tr>
<td>UnInitialized</td>
<td>Illegal</td>
<td>Sink State</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stimulus : ReleaseLock</th>
<th>Current state</th>
<th>Response</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnInitialized</td>
<td>Illegal</td>
<td>Sink State</td>
<td></td>
</tr>
<tr>
<td>CheckDockingStatus</td>
<td>Illegal</td>
<td>Sink State</td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>LockReleaseLock</td>
<td>No update</td>
<td></td>
</tr>
<tr>
<td>UnInitialized</td>
<td>Illegal</td>
<td>Sink State</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11: The state-box specification

The Clear-Box

The clear-box defines implementation procedures required to implement the state-box, and it can include other black-boxes recursively developed until the complete system
is constructed. A clear-box implements the state-box in any programming language, which supplies concurrency or sequential structures, such as sequence, alternation, and iteration. The box-structure space of a developed system can be seen in Figure 12. Clearly, the system is structured in a hierarchy.

![Figure 12: The box-structure space that realizes a system functionality](image)

Details of the clear-box are outside the interest of this report.

**Design and verification in BSDM** The box-structure development method implies the design and verification of systems by the expansion and derivation from one box to another: from a black-box specification to a state-box specification and then a clear-box specification for design, and the reverse for verification. Figure 13 depicts the steps of migrating from a box to another in a design step. A black-box is expanded to a state-box and then to a clear-box. For verification, a state-box is derived from a clear-box by eliminating implementation procedures, and then a black-box is derived by eliminating data states. To prove the correctness of a design, both the derived and the original black-boxes must be consistent. This can be established formally or by team reviews.

![Figure 13: The Box-structure development processes](image)

In [7], CSP and FDR (introduced in the subsequent section) are used to verify the consistency
between a black-box specification and a state-box specification automatically. FDR is used to verify determinism of a state-box specification automatically. The non-determinism in a state-box specification can be introduced by overlapping predicates (i.e., two predicates that may simultaneously evaluate to true). The behavioral equivalence between a black-box specification and a state-box specification is checked using the trace equivalent via FDR, after corresponding CSP processes are generated and checked for determinism automatically (both specifications must be deadlock-free and thus checking trace equivalence in this case suffices). Furthermore, CSP and FDR are used to verify behavioral equivalence between an original black-box specification and another derived black-box specification constructed through the abstraction process.

SBS and the ASD specification The ASD specification method is closely related to the SBS method. However, designing and verifying software components are slightly different. We elaborate on the relationship between the two specifications more thoroughly in Section 4.

2.3 Communicating Sequential Processes

Communicating Sequential Processes (CSP) [3, 5, 14, 16] is a formal specification language for describing the behavior and interactions of complex concurrent systems based on the theory of process algebras. CSP represents a mathematical framework to specify system behavior. This section describes briefly the syntax and semantics of CSP to the extent that is related to this work.

Syntax The BNF (Backus-Naur Form) syntax of CSP is given below, where $P$ and $Q$ denote the syntactic class of CSP process expressions, $E$ denotes the syntactic class of events (with $e \in E$ as a typical member), $A \subseteq E$ is a set of events, $ev \in Ev$ denotes a basic event name, $x \in X$ denotes a (data) variable, $v \in V$ denotes a data value, $p$ is a proposition and $c_0, c_1 \in C$ denote communication channel names.

$$
P, Q ::= STOP \mid SKIP \mid e \rightarrow P \mid c?x \rightarrow P \mid clv \rightarrow P \mid P \setminus A \mid P \sqcap Q \mid P \sqcup Q \mid P ; Q \mid P[[c_0/c_1]] \mid P \parallel A \mid P \downarrow p \mid P \uparrow v \mid Q
$$

$$
E ::= ev \mid ev.E
$$

Informal Semantics Next, we describe informally the intuition behind each CSP construct.

- The process STOP denotes a deadlocking process.
- The process SKIP denotes successful termination.
- Process $e \rightarrow P$ denotes event prefixing, i.e., it performs event $e$ and then behaves as $P$. Notable is the compound event constructed via the dot operator. Compound events are syntactic sugar, and can denote communication between a process and the environment through a shared channel. For instance the compound event $IMotor.Start$ denotes invoking a method $Start$ via a motor interface $IMotor$. Event $e$ might also have a certain structure to denote sending and receiving a value along a certain channel: $c?x \rightarrow P$ denotes receiving a value for $x$ from channel $c$ and $clv \rightarrow P$ denotes sending value $v$ to channel $c$.
- Process $P \setminus A$ denotes hiding occurrences of events in set $A$ from process $P$; to hide all such events they are renamed into the unobservable event $\tau$. 

11
• Process $P \square Q$ denotes external choice, where initial events of $P$ and $Q$ are offered to the environment and the actions present in the environment may determine the outcome of the choice.

• Process $P \cap Q$ denotes internal (non-deterministic) choice, where the process behaves as $P$ or $Q$ but the choice is made internally.

• Process $P \.; Q$ denotes the sequential composition of $P$ and $Q$. In such a sequential composition, first the events of $P$ may appear and upon termination of $P$, the events from $Q$ can take over.

• Process $P[\{c_0/c_1\}]$ denotes renaming: event $c_0$ becomes $c_1$ in $P$. We write $P[\{A/c.A\}]$ to denote renaming all occurrences of set $A$ in $P$; $a$ becomes $c.a$, where $a \in A$.

• Process $P \parallel_A Q$ denotes parallel composition of $P$ and $Q$ where they synchronize on events in the set $A$. In other words, events not in the set $A$ appear independently on either side, but events in $A$ should synchronize on the both sides of the parallel composition. The result of the synchronization is the same event, thus allowing for multi-party synchronization.

Interleaving parallel composition, denoted by $P \;||\; Q$, is a special instance of the general parallel composition with an empty set of synchronization events. In other words, $P \;||\; Q$ stands for $P \parallel \emptyset Q$.

Another variant of parallel composition is linked parallel composition denoted by $P[c_P \leftrightarrow c_Q]Q$; it intuitively means that $P$ and $Q$ synchronize with each other on channels $c_P$ and $c_Q$, respectively and then the result of the synchronization is hidden. Hence, for a fresh channel name $c$ not appearing in $P$ and $Q$, $P[c_P \leftrightarrow c_Q]Q$ stands for $(P[c_P/c] \;||\; Q[c_Q/c]) \setminus \{c\}$, where $\{c\}$ denotes the set of all communications on channel $c$.

• Process $P < | p | > Q$ denotes a conditional choice: it behaves as $P$ if $p$ is true, or $Q$ otherwise. We write $p \uparrow P$ to denote the process $P < | p \uparrow | > STOP$.

As syntactic sugar, we write $\square x: X(P(x))$ to denote replicated external choice over elements in set $X$. Similar notations are used to denote replication for $\cap$, $\parallel$, $\{x \leftrightarrow c_0\}$, $\;||\;$, and $\;\;$. In case of $\;\;$, the replication is done over elements of a sequence sequentially.

**Formal Semantics** In this section, we give the formal semantics of CSP (from [14]) constructs described below.

A CSP expression can be mapped into three semantic models: traces ($T$), stable failures($F$), and failures-divergences ($N$). The trace semantics of CSP defines the sequences of events that a certain process can afford. Note that due to the presence of (internal and external) choice and parallel composition, each process may afford more than one trace.

The formal definition of trace semantics of CSP is presented in Figure 14. Most of the definitions are self-explanatory. In order to define the trace model of parallel composition, we make use of the auxiliary function $t \;||\; u$ which calculates the parallel composition of two traces, whose definition is given below. $t \;||\; u$ is the smallest set satisfying the following constraints.

- $t \;||\; u = t \;||\; u$, $A$ $A$

\(^2\text{Written as } [[x \leftrightarrow c.x|x \leftarrow A]] \text{ in } CSP_M.\)
equal: they afford the exact same traces, denoted by traces. As far as the trace model is concerned, the two processes It only adds to the process UnInitialized addition to the process specified above: stimulus ends up in the process Initialized syntax; it prescribes the exact same pattern of stimuli and responses and in the case of the first malization in [7]): state "UnInitialized" can be formalized in terms of the following process (see details of the for-

\[
\begin{align*}
\text{traces}(\text{STOP}) &= \{\langle \rangle \} \\
\text{traces}(\text{SKIP}) &= \{\langle \rangle, (✓) \} \\
\text{traces}(e \rightarrow P) &= \{\langle \rangle \} \cup \{(e)^\ast t \mid t \in \text{traces}(P)\} \\
\text{traces}(e?x \rightarrow P) &= \{\langle \rangle \} \cup \{(e.v)^\ast t \mid v \in V \land t \in \text{traces}(P[v])\} \\
\text{traces}(P \square Q) &= \text{traces}(P) \cup \text{traces}(Q) \\
\text{traces}(P \cap Q) &= \text{traces}(P) \cup \text{traces}(Q) \\
\text{traces}(P \not\preceq \text{true} \triangleright Q) &= \text{traces}(P) \\
\text{traces}(P \not\preceq \text{false} \triangleright Q) &= \text{traces}(Q) \\
\text{traces}(P \parallel Q) &= \{s \mid s \in t \mid u, t \in \text{traces}(P) \land u \in \text{traces}(Q)\} \\
\text{traces}(P \setminus A) &= \{s \mid \exists e \in A (e)^\ast s \in \text{traces}(P)\} \cup \{(e)^\ast s \mid e \notin A \land (e)^\ast s \in \text{traces}(P)\} \\
\text{traces}(P; Q) &= \{s | s \in \Sigma^* \land s \in \text{traces}(P)\} \\
&\quad \cup \{(s)^\ast t | s \in \text{traces}(P) \land t \in \text{traces}(Q)\}
\end{align*}
\]

Figure 14: Trace Semantics of CSP [14]

- \( t \parallel_A u = \emptyset \) if \( t = (e)t' \) for some \( e \in A \) and \( u \neq (e)^\ast u' \) for any \( u' \),
- \( (e)^\ast s \in t \parallel_A u \) if \( t = (e)t' \) and \( u = (e)^\ast u' \) for some \( e \in A \) and \( s \in \text{traces}(t' \parallel_A u') \), and
- \( (✓) \in t \parallel_A u \) if \( (✓) \in \text{traces}(t) \land (✓) \in \text{traces}(u) \).

A process \( Q \) refines another process \( P \) in the traces model, written as \( P \sqsubseteq_T Q \) iff \( \text{traces}(Q) \subseteq \text{traces}(P) \). This means, every trace of \( Q \) is also a trace of \( P \). The following example applies the syntax of CSP to our running case-study and shows why trace models may be considered insufficiently discriminating for practical purposes.

**Case Study:** consider the state-box specification of DC component given in Section 2.2. The state “UnInitialized” can be formalized in terms of the following process (see details of the formalization in [7]):

\[
\begin{align*}
\text{UnInitialized } &= \text{IDockingCtr.initialize} \rightarrow \text{InitializeDevices} \rightarrow \text{Initialized} \\
&\quad \square \text{IDockingCtr.uninitialize} \rightarrow \text{Illegal} \rightarrow \text{STOP} \\
&\quad \square \text{IDockingCtr.ReleaseLock} \rightarrow \text{Illegal} \rightarrow \text{STOP} \\
&\quad \square \text{IDockingCtr.dockingConfirmed} \rightarrow \text{UnInitialized}
\end{align*}
\]

The above process translates the behavior specified by the state-box specification into CSP syntax; it prescribes the exact same pattern of stimuli and responses and in the case of the first stimulus ends up in the process Initialized, which is to be specified further. Consider the following addition to the process specified above:

\[
\text{UnInitialized}' = \text{STOP} \sqcap \text{UnInitialized}
\]

It only adds to the process UnInitialized the possibility of nondeterministically choosing for deadlock. As far as the trace model is concerned, the two processes UnInitialized and UnInitialized’ are equal: they afford the exact same traces, denoted by \( \text{traces}(\text{UnInitialized}) = \text{traces}(\text{UnInitialized}') \).
However, they may behave differently, i.e., process UnInitialized may initially decide to stop responding to any external stimuli. That is why we need finer models of behavior taking failures and divergences into account.

To distinguish these processes, one needs to consider what a process refuses to do along with what it can do. The set of events that a process may not accept at a stable state is called the refusal set, denoted by refusal$(P/s)$, where $P/s$ denote Process $P$ after a trace $s$ (the set of processes resulting from $P$ after taking the trace $s$, to be more precise). A failure is a pair $(s, X)$ which contains two elements: the first is a trace $s \in traces(P)$ and the second is a refusal set $X \in refusal(P/s)$ of the process after the given trace. The set $failure(P)$ denotes all failures of $P$. The semantic function $F[E]$ maps a CSP expression to its failures semantics ($F[E] = \mathbb{P}(\Sigma^* \times \mathbb{P}(\Sigma^*))$). The formal definition of failure semantics of CSP is presented in Figure 15.

$$
\begin{align*}
failure(STOP) & = \{((), Y) | Y \subseteq \Sigma^\prime\} \\
failure(SKIP) & = \{((), Y) | Y \subseteq \Sigma\} \\
failure(e \rightarrow P) & = \{((), Y) | e \notin Y\} \\
& \cup \{(x, Y) | x : V \land Y \land V = \{\}\} \\
failure(c?x \rightarrow P) & = \{((), Y) | x \in V \land (t, Y) \in failure(P)\} \\
& \cup \{(x, v) | x : V \land (t, Y) \in failure(P[v])\} \\
failure(P \cap Q) & = failure(P) \cup failure(Q) \\
failure(P \sqcap Q) & = \{((), Y) | ((), Y) \in failure(P) \land failure(Q)\} \\
& \cup \{(t, Y) | (t, Y) \in failure(P) \land failure(Q) \land t \neq ()\} \\
& \cup \{((), Y) | Y \subseteq \Sigma \land V \in traces(P) \land traces(Q)\} \\
failure(P) & = failure(P) \\
failure(P) & = failure(P) \\
failure(P \parallel Q) & = \{(u, Y) | Y \in traces(P) \land u \in s \parallel t\} \\
& \cup \{(t, Z) | (t, U) \in failure(P)\} \\
& \cup \{s, Y) | s \in \Sigma^* \land (s, Y \cup \{\} \in failure(P)\} \\
& \cup \{(s, t, Y) | (s', V) \in traces(P) \land (t, Y) \in failure(Q)\} \\
\end{align*}
$$

The property $F[E \cap E'] = F[E] \cup F[E']$ allows refining specifications gradually and incrementally (we elaborate on this in the subsequent sections).

A process $Q$ refines $P$ in the failures model, written as $P \sqsubseteq_F Q$ iff $traces(Q) \subseteq traces(P)$ and $failure(Q) \subseteq failure(P)$. Consider the following example. Let $P = (a \rightarrow STOP \sqcap b \rightarrow STOP)$ and $Q = (a \rightarrow STOP \cap b \rightarrow STOP)$ be CSP processes, and assume that the set of all actions comprises only $a$ and $b$. $P$ refines $Q$, though the reverse is not true since $P$ refutes only $\{}$ after the empty trace while $Q$ can additionally refuse $\{a\}$ and $\{b\}$ but not $\{a, b\}$. A process $Q$ refines $P$ if every failure of $Q$ is also a failure of $P$.

**Case Study:** the process Uninitialize refines Uninitialize' under failures model. The reverse is not true since Uninitialize' can additionally refuse to do any event at the empty trace.

Deadlock-freedom is a property that can be verified via the failures model. The process $DF = \cap_{e: \Sigma}(e \rightarrow DF)$ is the most nondeterministic deadlock-free process which can perform any trace and cannot refuse all actions in $\Sigma$. If a CSP process $P$ refines $DF$ in the failures model,
written as $DF \sqsubseteq_P P$, then $P$ is a deadlock-free process.

$$\text{failures}(DF) = \{(s, X)|s \in \Sigma^* \land X \subset \Sigma \land X \neq \Sigma\}.$$  

The failures/divergences model (N) extends the failures model to record the divergences of processes. A divergence is an infinite trace of internal actions ($\tau$, i.e., an infinite path of $\tau$'s). A process $P = a \rightarrow P$ after eagerly concealing $a$ (i.e., $\{a\}$) is a divergent process. When a process is engaged into a divergence, it is difficult to see what a process can do, and is assumed to do or to refuse anything and always diverges on any future trace. When a process up to a point diverges, one cannot rely on it doing anything. If two processes can immediately diverge, they are equivalent and completely useless.\(^3\) The set $\text{divergence}(P)$ contains the traces ($\bigcup_{s \in \Sigma^*} \{s\}$) after which $P$ can diverge ($\bigcup_{s \in \Sigma^*}$ denotes the trace $s$ and all its extensions, i.e., $s \tau$). When working with divergences, one needs strict sets of traces and failures. Such sets are defined as follow.

$$\begin{align*}
\text{traces}_+(P) &= \text{traces}(P) \cup \text{divergences}(P) \\
\text{failures}_+(P) &= \text{failures}(P) \cup \{(s, X)|s \in \text{divergences}(P)\}
\end{align*}$$

The formal definition of divergence semantics of CSP is presented in Figure 16.

\[
\begin{align*}
\text{divergences}(\text{STOP}) &= \{\}\n\text{divergences}(\text{SKIP}) &= \{\}\n\text{divergences}(e \rightarrow P) &= \{(e^\tau)t|t \in \text{divergences}(P)\}\n\text{divergences}(\text{true} \parallel Q) &= \text{divergences}(P)\n\text{divergences}(\text{false} \parallel Q) &= \text{divergences}(Q)\n\text{divergences}(P \parallel Q) &= \{(u^\tau)v|\exists s \in \text{traces}_+(P), t \in \text{traces}_+(Q), u \in (P \parallel Q) \land \Sigma^* \\
&\quad \land (s \in \text{divergences}(P) \lor t \in \text{divergences}(Q))\}\n\text{divergences}(P \setminus Y) &= \{(s \setminus Y)t|s \in \text{divergences}(P)\} \setminus \{(u \setminus Y)t|u \in \Sigma^\omega \land (u \setminus Y)\text{finite} \land \forall s < u.s \in \text{traces}_+(P)\}
\end{align*}
\]

Figure 16: Divergences Semantics of CSP [14]

A process $Q$ failures/divergences-refines another process $P$, written

$$P \sqsubseteq_{FD} Q \iff \text{failures}_+(Q) \sqsubseteq \text{failures}_+(P) \land \text{divergences}(Q) \sqsubseteq \text{divergences}(P).$$

Deadlock and livelock freedom of CSP models can be checked automatically via FDR. For a detailed treatment of the formal semantics of CSP, consult [14].

**Case Study:** we introduce the specification of two processes related to the docking controller component. We show that both processes are equivalent under the traces model and the failures model but not under the failures-divergences model. Firstly, consider the UnInitialized process specified earlier. Next, we modify the specification of the UnInitialized process such that a divergence in the empty trace is introduced. This is done by introducing a callback event that

\(^3\)For example, the process $P = (t \rightarrow P \parallel i \rightarrow g \rightarrow \text{SKIP})\setminus \{t\}$ and $Q = (t \rightarrow Q \parallel i \rightarrow \text{SKIP})\setminus \{t\}$ are equivalent under the failures/divergences model. The stable failures can be used to distinguish the two beyond the divergences.

15
denotes an internal notification from the sensor component of a detected docked platform in the Uninitialized state. Such a callback is consumed and ignored by the docking controller, so that no response is specified. The following is the specification of such a process:

\[
\text{UnInitialized}' = \text{IDockingCtr.initialize} \rightarrow \text{InitializeDevices} \rightarrow \text{Initialized}' \\
\text{IDockingCtr.uninitialize} \rightarrow \text{Illegal} \rightarrow \text{STOP} \\
\text{IDockingCtr.ReleaseLock} \rightarrow \text{Illegal} \rightarrow \text{STOP} \\
\text{ISensorCB.isDockedNow} \rightarrow \text{UnInitialized}' \\
\text{:}
\]

Then, we hide the internal callback from the process as follows:

\[
\text{UnInitializedDiv} = \text{UnInitialized}'\{\text{ISensorCB.isDockedNow}\}
\]

The process UnInitializedDiv refines UnInitialized under the traces model and the failures model; however they exhibit different behavior since the later can engage to a livelock and can stop responding to its environment (consider a faulty sensor sending endless notifications to the docking controller). The failures-divergences model can indeed distinguish UnInitialized from UnInitializedDiv.

**Software development through refinements** A software system can be developed in a sequence of stepwise refinements, starting with a simple specification and gradually refining it into a concrete implementation. Using the parallel composition operator in CSP, one can decompose and distribute a system into subsystems developed independently. For example, if \( P \parallel Q \) (with proper hiding) refines a specification \( S \), written as,

\[
S \subseteq_m P \parallel Q, \text{ where } m \in \{T, F, FD\},
\]

then we can develop the system by refining both \( P \) and \( Q \) independently. In case \( P \subseteq_m P_0 \) and \( Q \subseteq_m Q_0 \) then

\[
S \subseteq_m P_0 \parallel Q_0.
\]

In case \( P \) is a specification that is refined by a concrete implementation \( P_0 \), we can replace \( P \) by \( P_0 \). Specifications vary depending on the intended use. For instance, a specification can represent a component abstractly, a liveness property, or a safety property.

Consider Figure 2 again. \( S \) can be represented by a high-level specification that comprises more internal non-determinism. The specification \( S \) can then be refined by \( P \) and \( Q \) independently where \( Q \) is then refined by \( Q_0, Q_1 \) and \( Q_2 \). Ultimately, the composition of \( P, Q_0, Q_1 \), and \( Q_2 \) refine \( S \).

**A simple Buffer in CSP** We give a simple specification of a buffer and we use it along the paper. A simple N-Place buffer (defined in [3, 14]) can be specified as follows:

\[
\text{Gen}_B(N, in, out) = \text{Buffer},
\]

where \( \text{Buffer}=\text{[out} \leftrightarrow \text{in]}_{x<1..N} \text{COPY} \), and \( \text{COPY}=\text{in}?!x \rightarrow \text{out}!x \rightarrow \text{COPY} \). The process \( \text{COPY} \) represents a specification of one-place buffer. The process \( \text{Buffer} \) is a chain of one-place buffers linked together via the linked parallel operator. The specification of this buffer is adapted in ASD to detect cases when the buffer contains a maximum number of allowed elements. For this a special event called Queue-Full becomes visible, so that a user adapts the specification of the ASD models. Such ASD buffer specification is defined as follows:
where \( \text{Last} = \text{in}?x \rightarrow (\text{out}!x \rightarrow \text{Last} \sqcap \text{in}?x \rightarrow \text{Queue.Full} \rightarrow \text{STOP}) \), and \( \text{chase} \) is a compression function in FDR, by which internal \( \tau \) transitions are chased as much as possible. Note that queues in practice (using ASD generated code) are sufficiently big. In the appendix we provide an example specification that comprises the ASD buffer. In the rest of this paper, the process \( B_{\text{ASD}} \) denotes an ASD buffer, so that when we mention the queue overflows, we mean that the \( \text{Queue.Full} \) event becomes visible if an attempt is made to write to a full buffer.

An example specification in CSP of storing and retrieving ASD callbacks is given below. A process \( P = a \rightarrow cb \rightarrow P \) can store \( cb \) in \( B_{\text{ASD}} \) via the following combined process:

\[
P[cb/Qn.cb] \parallel B_{\text{ASD}}(QLen, Qn, Qt).
\]

If \( Q = cb \rightarrow Q \) consumes \( cb \), then the process becomes

\[
(P[cb/Qn.cb] || Q[cb/Qt.cb]) \parallel B_{\text{ASD}}(QLen, Qn, Qt).
\]

Clearly, such a combined process can cause the buffer to overflow for any arbitrary size \( QLen \) since \( P \) can send \( QLen \) consecutive elements without being strictly processed by process \( Q \).

3 ASD models

To develop an ASD component two models are required: an interface model and a design model. The models comprise ASD specifications, which use SBS as a main specification method (introduced in Section 4). An ASD model is a unit of modeling, verification, and code generation; required modification (due to changes of requirements, for instance) is often done solely in specification of models. ASD supports statistical testing [12] using the Compliance Testing Framework (CTF) specified through usage models. Usage models and statistical testing are outside the scope of this report.

![Figure 17: A collection of interface models and a design model related to the Docking Controller component.](image)
3.1 ASD interface model

The design process of any ASD component starts with creating an interface model which defines the external behavior of the component. This interface is checked for compatibility with its client design models. The interface is refined into a design model and possibly other new interface models. In turn, these new interfaces can be refined into concrete components following the same recipe. The ASD interface specification of a component only considers the behavior exposed to its environment while all other behaviors are abstracted away (e.g., interactions with used components). An interface model represents a protocol of interaction with respect to the component or system at lower-level, and often comprises non-deterministic behavior.

ASD interface models can also represent foreign components that are not developed in ASD (hardware, legacy code or handwritten code). From interface models, interface declarations (e.g., header files) as well as CSP models can be generated automatically.

Case Study: Figure 17 depicts an interface model (IComp) and its corresponding design model (Comp). The ILock interface model represents the behavior exposed by the lock device.

3.2 ASD design model

The essence of ASD as a design approach is to start with developing components first from their interface models and then to refine them into design models. The design model contains the concrete functional specification of a component, from which CSP code and implementation in a high-level language (e.g., C# or C++) can be generated. A design model represents a concrete design component which is always deterministic.

The design model of a component specifies an abstract dynamic behavior by means of state machines (automata) where data-related behavior is abstracted away. Data details are added (as external handwritten code) after the behavior of the component has been verified by model checking. Only events, component’s states and predicates are utilized in a specification of a design model.

Callbacks originated from used components, called in the sequel server components, are decoupled by a callback queue. From the implementation side, the queue of a component runs in a separate thread. If no callbacks is originated by server components, the component is always run in the thread of the caller.

4 Specification of ASD models

ASD uses the Sequence-Based Specification method [8, 6, 13] to specify the behavior of ASD interface and design models. The specification is extended and tailored to handle data and implementation details such as parameters with local and global declarations for source code generation. Such implementation-specific details are beyond the interest of this report. We mainly focus on the functional part of the specification.

4.1 SBS in ASD

ASD adopts SBS by starting the specification process of a model from the black-box which is extended to include transition predicates and canonical sequences (it is simply a Mealy machine represented in tables). Stimuli sequences are embedded in the analyzed canonical sequences, which provide a means to specify infinite stimuli sequences in finite partitions. This ultimately simplifies redesigning existing state machines to the corresponding ASD models. This also allows transformations from ASD specification to CSP (a generic algorithm is presented in [7]), high-level
source code (generated based on the state pattern introduced in [4]), and state machine diagrams. Notice that, the specification of ASD is extended to capture implementation details, such as return values and types of parameters in specified calls, for source code generation. Figure 18 depicts a screenshot of an interface model specified using the ASD specification via the ASD ModelBuilder. The specification contains two states (tables): the Uninitialized state with the empty canonical sequence, and the Idle state with the <ISensor.Initialized> canonical sequence. The Initialize stimulus is duplicated to denote a non-deterministic case (only in interface models, since design models should always be deterministic for source code generation).

The process of designing components in ASD excludes the steps of expanding the black-box specification to a state-box and clear-box. A stimuli might have more than one sequential responses executed until completion (return to the caller). The specification of a component is done by means of an interface model, which exhibits the external behavior of the component and a design model which includes the concrete functional behavior of the component. Both models must be consistent and checked via FDR after hiding all internals of the design model (the design model of a component and its used interfaces must refine the interface model of the component). Verification of ASD models is done compositionally and this is elaborated in more details in Section 5.

4.2 The specification in the ASD ModelBuilder

The ASD ModelBuilder provides tabs to facilitate the specification of models: the Client API tab defines implemented interfaces (i.e., methods invoked by clients which are also CSP events by which clients can synchronize) of the component; the Client Callbacks tab defines callbacks sent to client components; the Modeling Interfaces tab defines internal events which are not visible to client components; the Canonical Sheets tab defines a list of states; the State Variable tab defines predicates used in the specification of a model; the Tags tab defines a list of requirement identifiers of the component. Figure 18 depicts the way of specifying transitions among states and references to informal requirements (listed in the Tag tab). From the ASD specification, source code can be generated as well as state diagrams. Notice that there is no means to specify used interfaces of a component in the interface model. In the corresponding design model a user can include the used interface models (so that the responses to the used components are listed automatically to the user).

Figure 18: A specification of a force (on/off) sensor interface model
The followings are the main tips used in the specification process of components using ASD.

- Before modeling a component, we define the implemented, the callback, and the used interfaces of the component. They identify the boundary of the component. We start then by creating an interface model of the component, where only the implemented interfaces and client callbacks are considered. In the corresponding design model the used interfaces are further considered (the used interfaces are listed based on the included interface models of other used components).

- Based on the above interfaces, all input stimuli and all output responses of the component (the component alphabet) are listed and ready to be reflected in tables by the user. A stimulus or a response comprises a channel name and a method. In CSP a stimulus or a response is represented as a compound event (e.g., ichannel.method) while in implementation they are defined as the instance name and a method.

- ASD considers the set of stimuli of an interface model to include events with which a client can synchronize, and modeling events (explained below).

- ASD considers the set of stimuli of a design model to be the events with which a client can synchronize, callbacks from servers and synchronous return values.

- ASD considers the set of responses of a design model to comprise method invocations to used components and callbacks sent to clients’ queues. The set of responses of an interface model excludes any method invocations to the used server components. The response events, on which a client and server can synchronize, are stimuli of used ASD interface models and responses of the ASD design model. The responses are extended with four special responses: Illegal, Blocked, Null, and IF.NullRet (addressed in 4.2.1). If a design model uses an interface model (included via the ASD ModelBuilder), all responses are listed automatically.

- Each state in the state machine of the model is represented as a table where all possible stimuli are listed. Each row (called an ASD rule case) consists of a single stimulus and its corresponding response(s). Responses are filled based on informal requirements or a state machine in case it exists a priori. The main essence here, however, is to start with SBS and ultimately to end up in a specification that can be seen as a state machine. Starting from SBS allows users to understand their software design and derive new requirements while they construct their specification. This is because all stimuli are listed in each table and for each stimulus the corresponding responses have to be filled in.

- While responses are assigned, new tables that represent new states can be created. This is in case responses lead to new states. For this, new partitions with their canonical stimuli sequences are created automatically.

- In case responses are conditional, predicates can be used to indicate this.

- In an ASD interface model, internal interactions that are hidden from clients and use the interface are specified as modeling event stimuli. Client callback events can be specified as responses to these events to model the case where the client is informed about something that had internally happened. The user adds the suffix “INT” to the channel names of such modeling events for readability. Such modeling events are not part of any synchronization and hence they are triggered as soon as the system enters the state where they are specified. They are the main sources of interleaving.
Data can be passed as parameters (data independence is assumed, i.e., data cannot influence the control behavior). Data processing and computations must be done by using handwritten code.

After all entries in tables are filled, CSP or C# code as well as a state machine diagram can be generated.

**Case Study:** Figure 18 depicts the ASD specification of the interface model of the force sensor.

### 4.2.1 Special responses

To make the specification total, the list of user-defined responses is extended with special responses: **Illegal**, **Blocked**, **Null**, and **IF_NullRet**. The **Illegal** response is used when invoking a stimulus at a state is illegal. Basically, it is an event, which leads to a sink state that allows detecting unexpected illegal calls. This can be verified by formulating a CSP property which can be checked via traces refinement as follows.

$$\text{STOP} \sqsubseteq T \text{Design}\setminus \{\Sigma/\{\text{Illegal}\}\},$$

where “/” denotes set difference and \(\text{Design}\) denotes the CSP process under verification. So, if the **Illegal** event is visible in any trace of the combined process \(\text{Design}\) (notice that all events are hidden except **Illegal**), FDR gives counterexamples. In the above property, the **STOP** process is used in the refinement check instead of the **SKIP** process since the ASD processes do not terminate. Via this property, one is only interested to find a trace that contains the **illegal** event. Furthermore, hiding all events except the **illegal** event facilitates applying the compression functions supplied by FDR for verifying complex models with a large state space.

![Figure 19: The usage of the Illegal event](image)

To clarify the significance and usages of the **Illegal** event, consider the example below. In Figure 19a., the client in state 1 can issue either the Start or the Stop call to its server (the choice between the two options is an internal choice, i.e., solely determined by the client). In practice, if the client issues the Stop call, the system may fail since the server in state 1 is only willing to receive the Start call. Using model checking, both the client and the server can communicate in state 1 without deadlock (Stop is blocked and verification properties are required to check the case). In this case assigning the **Illegal** response to the Stop stimulus at the initial state of the server (Figure 19b.) would detect such an illicit call and hence either the client or the server specification has to be adjusted. In ASD the **Illegal** event is mainly used as a response to server callback events: to check race condition scenarios, for instance. By using the **Blocked** response, no corresponding CSP code or source code is generated for the attached stimulus. The **Blocked** response denotes that the stimulus can never happen. The **Null** response denotes no response:
consuming a callback, for instance. The *IF.NullRet* (where *IF* denotes the interface name) is used for void calls. The *IF.NullRet* can allow or block new calls to be processed. In the specification of a component, if the *IF.NullRet* is issued as a response to a client call, the component can process a new client request after a previous call is completely processed and returned. A user can postpone the *IF.NullRet* (hence the component blocks new requests if any) to process internal solicited callbacks. Therefore, a component can process a request with all its internal callbacks before any new external event is handled. Special locks and thread monitors are used in the implementation of components, of which the details are outside the scope of this paper.

### 4.3 Models of interaction among ASD components

In this section, we informally describe the run-time semantics of ASD components. The main aim is to give the reader an overview of how ASD components interact with one another in practice. The subsequent section is devoted to introduce the formal CSP semantics corresponding to the run-time semantics.

Client and server ASD components can in practice exchange messages using synchronous or asynchronous calls. For the majority of ASD design cases, a client component always calls the server component synchronously. The non-durative action means that, if a client invokes a server method, the client blocks other external requests until the server processes the call and the call is returned with a value (or possibly void return) to the client, see Figure 20a. The server can report something back to a client via synchronous return values or via asynchronous callbacks sent to the queue of its client. The durative action means that the server can issue asynchronous callback events after a non-durative action, see Figure 20b. Such callbacks are sent always to the queue of the client component.

**Figure 20:** a. Non-durative actions b. Durative actions

**ASD Run-to-completion and monitor semantics** The run-to-completion semantics means that when a stimulus is invoked in a state, its corresponding responses are processed completely in the specified order and all predicates are updated before the transition is made to a next state.

The monitor semantics implies that a new client request is delayed until a previous request has been completely processed (the non-durative action). In other words, components are non-reentrant and thus, only one client request is processed at a time. The diagram in Figure 21 explains the Monitor semantics of a server component that has two clients and one server components. It also clarifies the usage of the *IF.NullRet* event. Both run-to-completion and monitoring
semantics are supported in any ASD component. Thread locking and semaphores are used in the implementation, of which the details are outside the interest of this report.

5 Composing and verifying components in ASD

5.1 The structure of the translation into CSP

In ASD a system is developed and verified in steps in a component-wise manner. In each verification step, the following CSP models are generated and combined together (CSP processes are generated from ASD models as described in Section 2.3): component interface model, component design model, a process that represents the environment of the design model, used interface models, and possibly the queue of the design component in case callback events exist. Using our case study, we demonstrate an example for clarification.

Case Study: consider the setup of models depicted in Figure 17. Let \( \mathfrak{A}(ASD_{\text{model}}) \) denote the result of the algorithm (presented in [7]) that generates a CSP model from a corresponding ASD specification \( ASD_{\text{model}} \). Let \( A \) denote a set of DC client requests which are stimuli in DC and responses in SC, \( cNR \) be the corresponding NullRet event that unlocks the requests of SC. Let \( R \) denote the set of DC responses which are stimuli in ILock and ISensor interface models, \( sNR \) is the server NullRet corresponding to \( R \), and \( CB \) is the set of callback events originating from ISensor(\( CB' \)) and ILock(\( CB'' \)) to DC queue. Let \( EP = \Box \ a: A (a \rightarrow cNR \rightarrow EP) \) denote the environment process of DC and \( Buf = B_{\text{ASD}}(QLen, Qn, Qt) \) be an ASD queue specified in Section 2.3. A composed process \( DP \) can be constructed as follows.

\[
DP = (\mathfrak{A}(DC)[[\Box B/Qt, CB]] \parallel UesdIF) \parallel Buf) \parallel EP,
\]

where \( UesdIF = \mathfrak{A}(ISensor)[[\Box B'/Qn, CB']] \parallel \mathfrak{A}(ILock)[[\Box B''/Qn, CB'']] \parallel A_{\{\Box cNR\}}, \) and \( CB', CB'' \subseteq CB \). In ASD such a combined process is checked for deadlocks and livelocks using FDR. Notice that modeling events are not part of any synchronization with any party in the combined model, so that they are triggered spontaneously.

Figure 21: A Monitor semantics. MessageB is blocked until a previous call is returned.
The following property checks whether $DC$ and its used interfaces refine the specification $IDC$:

$$\forall(IDC) \sqsubseteq_{FD} DP \setminus R \cup \{sNR\} \cup \{|Qn, Qt\}.$$  

The following property checks whether the queue of the combined process overflows.

$$STOP \sqsubseteq_T DP \setminus \{\Sigma/\{Queue\_Full\}\}.$$  

In ASD the $DP$ process is also checked for deadlock and livelock freedom. After verification of a component, it is guaranteed that the client component in practice can call the concrete server components without any deadlock or illegal behavior. The client and server components never exhibit illegal calls in any execution scenario, and their implementation assures their proposed functional behavior. In the appendix we provide an example specification of a client and a server component, which can be directly verified in FDR.

5.2 Importance of eager and lazy abstractions

In ASD interface models, a user can select the abstraction type of modeling events to specify the type of internal behavior: inevitable or optional. For simplicity, we assume that callbacks are responses to modeling events, in what follows. With the inevitable type, callbacks are always expected to be fired, while with the optional type, callbacks might or might not be fired (lazy abstraction where a callback is nondeterministically offered - in combination with a STOP event to keep the interface silent - to the client). The optional type denotes firing callback events that might or might not be issued during the execution of the system: a sensor `movementIsDetected` callback, for instance. A system simply deadlocks if a client progress is completely reliant on the arrival of an optional callback.

To clarify the above, consider the following specification. Let $SP = m \rightarrow cb \rightarrow SP$ denote a CSP process of an interface specification of a server component. If event $m$ (modeling event) is eagerly abstracted (standard hiding) then $cb$ (client callback) is always offered to the client. That is,

$$Eager_{\{m\}}(SP) = \tau \rightarrow cb \rightarrow SP.$$  

However, in case $cb$ is optional, $m$ should be lazily abstracted. To realize this, a special process $CHAOS$ (a process which may or may not do any action from a given set) is run in parallel with $SP$ sharing the set of modeling events and then hiding them as usual. That is,

$$Lazy_{\{m\}}(SP) = (SP \parallel_{\{m\}} \{CHAOS\}) \setminus \{m\},$$

where $CHAOS_E = \cap_{e:E} ((e \rightarrow CHAOS) \cap STOP).$

Thus, the resulting $SP$ process after the lazy abstraction of $m$ is

$$Lazy_{\{m\}}(SP) = \tau \rightarrow ((cb \rightarrow SP) \cap STOP).$$

Therefore, if the progress of a client component is completely reliant on the arrival of the optional callback $cb$, the client deadlocks.

**Case study:** in the sequel, we introduce the concept of optional and inventable modeling events to our case study. In the Sensor interface model (see Figure 18), we have specified the modeling event `ISensorINT.IForceDetected` as an optional event, which denotes an internal behavior representing the case of when the on/off sensor becomes active due to a force caused by a docked platform. So that the `ISensorINT.IForceDetected` is lazily abstracted, and hence the sensor might
or might not offer the ISensorCB.isDockedNow callback to its client (the callback becomes optional).

On the other hand, as an example of using the inevitable type, the Movement Controller can use a handwritten component that implements a timer to insure that initializing its motor will not exceed a predefined time. The Movement Controller will in this case consider the timeout callback to be inevitable. If such a timeout callback is optional and the movement controller is completely reliant on its arrival, then the movement controller deadlocks (in case the motor is faulty and does not respond to the initialize request).

Notice that, in some practical applications, an ASD interface model might comprise complex behavior and might include a number of optional modeling events, then it is practical to compose in parallel the generated process (of the interface model) once with a global CHAOS process, instead of adding a number of CHAOS processes incrementally.

5.3 Refinements by set inclusion in FDR for incremental development

The refinement models supported by FDR allows development of software gradually and incrementally. A user can specify an interface specification of a future design which can be used by third-party client components. This specification forms a contract, promises what the clients should expect from the system under development, and comprises expected services of the future design. For example, a subsystem can non-deterministically offer $b$ and $c$ services to its clients as follows:

$$SP = a \to b \to SP \square a \to c \to SP.$$  

Client components can verify their behavior with respect to the used interface $SP$ which can be in turn refined gradually in steps. A user can develop the first concrete service of $SP$ as follows:

$$SP' = (\tau \to) a \to (\tau \to) b \to (\tau \to) SP'.$$

In ASD terms, a design model and used interface models are created with detailed internal behavior $\tau$’s. $SP'$ refines $SP$ and code can be generated and deployed to the system. The next step is to develop the second concrete service. This can be established by extending the specification and the internal details of $SP'$ as follows (by extending the specification of existing models or adding new used interface models, hence $SP''$ is an alternative for $SP'$ and not a refinement over it):

$$SP'' = (\tau \to) a \to (\tau \to) b \to (\tau \to) SP'' \square (\tau \to) a \to (\tau \to) c \to (\tau \to) SP''',$$

which is still a valid refinement of $SP$. The above can be considered as one increment. The next increment might comprise extending both the clients and the developed design components with new functionalities. Notice that ASD interface models are often divergence-free, so the refined services must also be divergence-free.

6 Observations and discussions

In conventional design methods, model checking can be incorporated in software development for verification after a design has been made and possibly after implementation; however, extracting a correct abstract model that reflects the actual behavior of what is (going to be) implemented is considered to be a complex task. The formal mathematical models are created manually or semi-automatically, so that changes are required in both the design and its corresponding formal model separately. Fortunately, the call-and-return and SBS styles employed in ASD facilitate incorporating model checking in software development, and allow fully automatic transformations to formal models. Using SBS a designer is required to think abstractly (in an action-oriented
manner), and design details including data computations are deferred to later stages. In ASD, changes are done solely on models, and this can provide an enhancement of cost, quality, and time in software development compared to conventional methods. However, designing software using ASD is not an easy task, especially for designers who lack the action-based mind-set, though ASD hides formal details from end-users. A careful analysis and investigation of a design a priori applying ASD are required before designers run into problems unpredictably.

7 Development of the case study

7.1 Development of software components

The development of the system is started by identifying the components and their responsibilities. We take the approach of designing software using box-structure but in an ASD way (i.e., thinking abstractly as inspired from the black-box specification without constructing any state-box or clear-box, or putting any data-processing into account at the starting step). At first the system was seen as a black-box (represented by an ASD interface model and later a design model) with highly abstract states that identify the overall system states. This leads to identify the top-level component (System Controller) with corresponding abstract stimuli and responses that only affects such abstract states. During this step other black-boxes (represented by ASD interface models, i.e., not design models) emerge, and become subject to further decomposition. The new black-boxes are treated independently resulting in identifying both the Movement Controller and the Docking Controller with their detailed behavior. Figure 22 depicts the steps of decomposing system components in a hierarchy.

As soon as the decomposition of components has been achieved and the responsibilities are identified, one can start with implementing concrete components by constructing ASD models implemented and verified compositionally. We have chosen to implement our concrete components bottom-up as depicted in Figure 23.

7.2 Results

The requirements given in Section 1 are reflected in the SBS's of the components. The specification and verification of our case study took about one person-week cumulative effort, and during verification more than 50 design defects have been identified. These defects emerged due to race conditions, deadlocks, livelocks and missing transitions and states. Notably, when a defect is fixed,
others easily and unpredictably emerge; these defects are detected by a push button in few seconds using model checking.

Since verification and development of the components are done in a compositional way, the state space explosion is circumvented. The interface model of the System Controller represents the whole system for other clients which use system functionalities. New functionalities and components can be added and a new increment can start: for instance, adding a light component, handling hardware fault scenarios, and timers for requests’ timeouts.

8 A discussion on model checking in ASD

During the development of our case study we encountered some ASD model-checking-related issues such as the queue-full and divergences. In the reminder of this section, we detail their causes as well as the way to circumvent them when future system designs are considered.

8.1 Queue-full identification and prevention

The Queue-full is encountered when a server continuously sends callback events to its client queue and the client does not strictly process them. In general, callbacks are generated from a server interface model either as direct responses to client calls or as responses to modeling events which denotes internal processing. In both cases a client can block or allow other new requests to be processed before callbacks are received (via NullRet or return values). The callbacks that are fired as a direct consequence of a client call where the client is strictly waiting for them (by blocking new requests) are called the solicited callbacks. Callbacks that are responses to modeling events are called internally-stimulated callbacks regardless whether the client is strictly waiting for them or not. If a client allows new requests (that cause future callbacks from the servers) to be processed while callbacks related to old request are intact in the queue, the queue simply overflows.
8.1.1 Treating solicited callbacks

Such callbacks cause a queue of a client to overflow if the client is not strictly alternating. This means that a client at a state can either process a callback or a new client request that causes the servers to send other callbacks. A simple CSP specification that illustrates the queue-overflow is presented in the appendix.

To circumvent this, the client specification must be strictly alternating with respect to a client request. A client blocks other new requests until a solicited callback is received. Figure 24 shows a basic interaction model between a client and a server where the client blocks its clients until it receives the solicited callback from its server.

![Figure 24: A strictly alternating client](image)

**Case study:** Both the initialize and uninitialize requests and their callbacks in each component must be strictly alternating (otherwise the queue becomes full of initialized and uninitialized callbacks). That is, the uninitialize request is not allowed to be processed by any controller until all lower-level components have processed the initialize request completely and have sent their corresponding initialized callback successfully. For this, each component will issue the NullRet event as soon as the initialized callbacks are received from its server components. The same applies to the uninitialized callback.

To clarify the above, consider the specification of the Docking Controller depicted in Figure 25. In rule (row) 6, the Docking Controller can receive the initialize request from the system controller, and as a response to the call it initializes both the sensor and the lock components. The call is not returned to the System Controller and the Docking Controller transits to the *CheckDockingStatus* where it expects a solicited callback from the sensor indicating the docking state. When the solicited callback is received (in either rule 18 or 19), the Docking Controller sends a callback event to the queue of the System Controller and allows the initialize call to return to the System Controller (by issuing the NullRet event). The System Controller continues with its next activity (processing the callback and initialize the Movement Controller). Notice that we assume here a reliable hardware that always responds with a callback. In practice hardware might not respond with the expected callback due to some internal faults, so the system will simply deadlock in this case. For this ASD supplies a timer for timeout monitoring. The details of the timer are outside the scope of this report.
8.1.2 Treating internally-stimulated callbacks

Internally-stimulated callbacks are always specified as responses to modeling events in interface models. Modeling events are not part of any synchronization in the combined model, so that they can be triggered as soon as the model enters a state where they are specified. The Queue-Full is encountered if the interface is allowed to continuously send callbacks to the client's queue while the client does not strictly process them. To circumvent this, predicates or transitions to new states where these modeling events are blocked should be maintained. So, predicate are set when callbacks are sent to the queue. Predicate are reset when a special call indicates that the client has successfully processed the callback. It is simply a handshaking between the client and the server where the client acknowledges receiving the callback to the server by a synchronous call (in the case study we always confirm a server component of receiving such callbacks, so that the server component can send the callback if needed).

Below is a specification of a client and an interface model of a server component that causes the queue of the client to overflow.

Let $SP = m \rightarrow cb \rightarrow SP$ denote a server interface that sends a callback $cb$ as a response to a modeling event $m$. Let $CP = cb \rightarrow CP$ be a client component that is intended to consume the callback. Combining these processes with the ASD queue $Buf$ causes $Buf$ to overflow for any arbitrary queue-size $n$. That is,

$$CP[cb/Qn.cb] \parallel SP[cb/Qn.cb]\parallel Buf$$

causes $Buf$ to overflow.

This requires the specification of the server component to be adjusted as follows:

$$SP(p) = (m \rightarrow cb \rightarrow SP(true) \downarrow p=false \uparrow ack \rightarrow SP(false))$$

and the client as follows:
\[ CP = cb \rightarrow ack \rightarrow CP. \]

Hence, the combined process can have the following form:

\[ (CP[[cb/Qt.cb]] \parallel SP[[cb/Qn.cb]] \parallel \{\{Qn,Qt\}\}) \parallel Buf. \]

Modeling events cause interleaving, so that internally-stimulated callbacks can be seen in each state of the client design model. In future versions of the ASD ModelBuilder, this will be circumvented by introducing the concept of yoking, where only a maximum number of certain callbacks are allowed in the queue (but in implementation callbacks are allowed as many as possible at a time). Furthermore, a user can specify callbacks of type singleton events which indicate that only one instance of a callback is processed at a time, so a new instance of the same callback is simply discarded if there is another instance in the queue waiting for processing (also in implementation). In future ASD ModelBuilder any callback event will be processed before any external client request (by a so called DPC barrier process).

### 8.2 Benign divergences

Another commonly encountered problem is concerning divergences, which we discuss in the remainder of this section. The divergence check is used in ASD to guarantee that two processes (the interface versus a design model and used interface models) exhibit the same external behavior. The divergence in ASD models might be introduced if the refined models comprise a loop of internal behavior which is not visible at the interface (see Figure 26a.). For instance, a design model receives a callback from its server interface model and based on this callback the design model calls the same or other server interfaces without reporting callbacks to the client of the design model in between (i.e., the behavior is not visible at the interface model). As can be seen in the system developed as the case study, the Docking Controller can apply the lock immediately when it receives the \textit{IsDockedNow} callback from the sensor without reporting the docking status as a callback to SC (see Figure 26.a). In this case, the sensor \textit{isDockedNow} callback and the \textit{applyLock} call are not visible at the interface IDC so the behavior is considered as a divergence.

![Figure 26: a) Divergences caused by hiding internal actions b) Divergences caused by hiding other API communication channels of other components](image-url)
To circumvent this problem in practice, users can increase the number of counterexamples produced by FDR (maximum 100) and investigate all cases of divergences manually. In case such divergences are legal, users can choose the failures model to proceed with verification and to skip the divergence option. This is because most of practical systems are divergence-free. In general, the failures model lets us see the details of behavior after possible divergences.

Another suggestion is to break the tau loop by introducing a visible event (a client callback) on both the interface and the design models. In our case, we forward the IsDockedNow callback to SC and hence it becomes visible at IDC. So that our system formally becomes divergence-free although this was expected before.

A divergence might also be introduced in case a server interface is used by more than one client (see Figure 26b. and consider the divergence refinement check of interface model IX after hiding all Y interactions). The same solutions apply to this case.

8.3 Significance of verification properties

Properties allow a user to specify in a direct way what is to be expected from the system. In this way more design flaws can be detected than using the built-in checking facilities. For example, a model can be deadlock-, livelock-, and illegal-call-free but, at the same time, not provide its intended behavior. A model checker can capture a subset of the intended behavior due to mistakes in the formal specification by a user. Few live scenarios in a model might restrain detecting the reachability of other scenarios. For example, consider the following specification of an interface model:

\[ SP(p) = (a \rightarrow cNR \rightarrow SP(false)) \Box (m \rightarrow cb \rightarrow SP(false) \downarrow p = true \downarrow STOP) \], where \( p \) is initially false.

In this example the client that uses this interface might be deadlock-, livelock- and illegal-call-free since it is only allowed to synchronize on \( a \) (in that case the client should not be reliant on \( cb \) at any state). However, in slightly more complex designs, it may go unnoticed that action \( cb \), although specified for some reason, is never reachable from the initial state and this may cause further problems, if not checked at this stage. Such an anomaly can be easily detected by formulating a dedicated property. For example, the property

\[ STOP \sqsubseteq_T DP\setminus(Events/\{cb\}) \]

causes FDR to detect all traces that contain \( cb \) in the combined model \( DP \) (counterexamples can be increased to 100 maximum). In case this property is successfully verified, this indicates that \( cb \) is not part of any trace of the model. Therefore the designer becomes aware of the this fact and specification can be adjusted accordingly. Checking this type of mistakes by safety properties is not valid since safety properties hold in any model that does nothing.

Notice that safety and liveness properties can be checked via traces and failures models. Safety properties can also be checked via the failures model but this adds unnecessary complexity to the specification of the property. This can become an obstacle for complex properties.

9 Conclusions

In this report an introduction to ASD as a component-based technology is presented. We elaborated on the formal and informal details that form the main concepts of ASD. We explained types of ASD models specified using the sequence-based specification method. A case study of a patient trolley controller and the realization of its functionality using ASD is detailed throughout the report. Using ASD software design is performed in increments, each of which controlled mainly using model checking. ASD provides a formal, structured, and systematic approach to software
development resulting in a decrease of time and cost efforts, and an increase of quality of software systems. We introduced commonly encountered issues when modeling components behavior in ASD, and we supplied practical solutions to circumvent them. We are currently applying ASD on various projects within our industrial domain, reporting about our gathered results and experience coupled with further extensions of the ASD method.

Acknowledgments

We would like to thank Guy Broadfoot and Philippa Hopcroft for their comments and suggestions. Also thanks go to Jan Friso Groote for his valuable comments. The following people, in alphabetical order, have helped us get more insight in the application of ASD to industrial cases and thus are gratefully acknowledged: Paul Alexander, Bert Folmer, and Tom Fransen, Harry Kuipers, Ron Swinkels, Jacco Wesselius, Marco van der Wijst.

References


APPENDIX

An example of composing processes in ASD

We give an example specification of a simple client (CP) design model and server (SP) interface model, each of which comprises a single state. Let \( a \) denote a client request, a call to the server is denoted by \( r \) which is a response in the client and a stimulus in the server, and \( cb \) is a callback event originated from the server to the queue of CP. Let \( EP = a \rightarrow cNR \rightarrow EP \) denote the environment process of \( CP \), \( CP = a \rightarrow r \rightarrow sNR \rightarrow cNR \rightarrow CP \) \( \Box cb \rightarrow CP \) denote the client process (design model), and \( SP = r \rightarrow cb1 \rightarrow cb2 \rightarrow sNR \rightarrow SP \) a server process (interface model of a server). The combined process \( DP \) takes the following form:

\[
((CP[[cb/Qt.cb]] \parallel SP[[cb/Qn.cb]]) \parallel Buf) \parallel EP.
\]

Let \( ICP = a \rightarrow cNR \rightarrow ICP \) denote the client interface specification (interface model of \( CP \)). The following property checks whether the combined process \( DP \) refines \( ICP \):

\[
ICP \sqsubseteq_{FD} DP \setminus \{r, sNR\} \cup \{|Qn, Qt|\}.
\]

The above-given refinement does hold due to a deadlock caused by the queue-overflow.

Below is the corresponding specification of such processes, which can be verified directly in FDR2.

```plaintext
channel a, b, cb, r, cNR, sNR, QUEUE_FULL
B_ASD(N, in, out) =
  let
    external chase
    LAST = in?x -> (out!x -> LAST [] in?x -> QUEUE.Full -> STOP)
    COPY = in?x -> out!x -> COPY
    Buff = [out<->in] x : <1..N-1> @ COPY
    within chase(LAST [out<->in] Buff)
  CBEvents = {cb}
channel Qn, Qt : CBEvents
CBQLen = 10
Buf = B_ASD(CBQLen, Qn, Qt)
EP = a -> cNR -> EP
CP' = a -> r -> sNR -> cNR -> CP'
  []
    cb -> CP'
SP' = r -> cb -> sNR -> SP'
SP = SP'[[x <= Qn.x | x <= CBEvents]]
CP = CP'[[x <= Qt.x | x <= CBEvents]]
UnQDP = CP [[{r,sNR}]]SP
DP' = UnQDP[{|Qn, Qt|}]Buf
DP = DP'[[{{a,cNR}]]]EP
ICP = a -> cNR -> ICP
assert ICP [FD= DP \diff(Events,{{a, cNR}})]
assert STOP [T= DP \diff(Events,{{QUEUE.Full}})]
```

To circumvent the queue-overflow in the above example, the specification of \( CP' \) is replaced by another strictly alternating process as specified below.

```plaintext
CP' = a -> r -> sNR -> CP'
  []
    cb -> cNR -> CP'
```