Model based design-space exploration for Chuck Exchange

Vyas, A.

Award date:
2015

Model based design-space exploration for Chuck Exchange

Avanti Vyas
Student Id: 0871539
Embedded Systems
a.vyas@student.tue.nl

University Supervisor
Dr.ir. Jeroen Voeten
Department of Electrical Engineering, TU/e
J.P.M.Voeten@tue.nl

Company Supervisor
Mr. Wouter Tabingh Suermondt
Principal Functional Architect, ASML
wouter.tabingh.suermondt@asml.com

August 31, 2015

Abstract

ASML is the world leader in the production of Lithography Systems for semiconductor industries. Lithography is one of the processes involved in making integrated circuits (ICs). ASML designs and manufactures these complex systems which process silicon wafers to eventually produce ICs. Silicon wafers are processed by executing so-called sequences of actions, where each action carries out an individual processing step. This MSc project focuses on one of the critical sequences which synchronously exchanges the chucks used for measurement and exposure. During the Chuck Exchange two Synchronization Controllers have to act upon a single resource, potentially introducing a race condition.

The goal of this work is to design a robust solution, in which race conditions are always avoided. To this end a formal Functional Specification for Chuck Exchange is constructed by reverse engineering the existing Implementation Specification and Execution Framework that implements the *Chuck Exchange Sequence*. In addition, a detailed model of the Execution Framework is created to analyze the potential race condition. Based on an analysis done, an Implementation Rule is derived that guarantees the absence of any potential race conditions. An improved Implementation Specification is designed, based on the Implementation Rule. This robust Implementation Specification is shown to respect the Functional Specification. In addition, it is shown that the existing Implementation Specification and the robust Implementation Specification are equivalent in the sense that they produce the same Gantt chart which implies that the robust Implementation Specification leaves the performance intact. Consequently this Implementation Specification is implemented in the detailed model of the Execution Framework to increase the evidence of correctness of the solution. The solution proposed executes the *Chuck Exchange Sequence* in a robust manner, having the potential to improve system availability.

Keywords: Chuck Exchange, Functional Specification, Implementation Specification, Execution Framework, POOSL model.

Acknowledgement

This project would not have been possible without the support and guidance of my supervisors, colleagues, friends and family. I would like to extend my sincere gratitude towards all of them.

First of all, I would like to thank my university supervisor Dr. ir. Jeroen Voeten for guiding me throughout this project. Every discussion I had with him helped me build up my skills and grow. His critical evaluation of my work kept me on my toes and helped me improve myself. He introduced me to new concepts which were beneficial in building up a strong technical framework while working on the project which otherwise I wouldn't have learnt easily. I would like to specifically express my gratitude towards him for patiently reviewing my report.

I owe a great deal to my ASML supervisor Mr. Wouter Tabingh Suermondt. The information within ASML that was related to my project was always readily made available by him. By sharing his wisdom and experience he guided me throughout the project. He also helped me in keeping up with the schedule. He was always available whenever needed. I would also like to thank Mr. Rob Jagt for dedicating his precious time whenever required in solving all my doubts and explaining the essentials. This project would not have been accomplished without his constant help and knowledge sharing. I would also like to thank Mr. Jack Bombeeck and Mr. Richard Niessen for solving all my queries and providing the required information as and when needed.

I would like to show my sincere gratitude to all my colleagues at HTC for their valuable inputs especially during my presentations. I would like to thank Miss Fangyi Shi and Mr. Yuri Blankenstein for helping me with the tooling. I would also like to thank all my friends and colleagues in Veldhoven for creating a friendly ambience to work.

I would not have been able to accomplish my work without the emotional support of my friends (especially Saurabh, Aniket, Ketan, Ashwath, Ajay, Richard, Anupama and others) in Netherlands. They constantly encouraged me and helped me in giving a sincere peer review. They supported me emotionally like a family during my stay in Netherlands.

I owe this project to my mother Dr. Sarita Vyas who patiently encouraged me to continue in times when I felt low. I take this opportunity to thank my father Dr. Sunay Vyas for always supporting me in all my decisions and my little brother for always being there for me.

Last but not the least I would like to thank my dear friend Aniket Lewarkar for his unconditional support since I came to the Netherlands and even before.

Table of Contents

Table of Figures

Chapter 1

1.1 Introduction

ASML is the world's leading provider of Lithography Systems for the semiconductor industry. Lithography is defined as the process used to create circuit patterns in multiple layers on a silicon wafer that eventually produces an integrated circuit (IC). ASML designs and manufactures these complex Lithography Systems which are critical to the production of ICs.

Typically, a Lithography System involves creating a circuit pattern on the silicon wafer by exposing the pattern and the wafer with a light source. The image of the pattern is focused through a projection lens onto the silicon wafer. This requires measuring of the wafer before exposing it so that the pattern is printed at the right position to eventually have multiple layers of pattern on the wafer. Thus, measuring and exposing of the wafer are the two important processes involved in a Lithography System.

1.2 Problem Description

In order to achieve high performance, the measurement and the exposure are performed independently in the TWINSCAN Lithography Systems of ASML. A TWINSCAN Lithography System has a wafer stage which employs a dual stage concept (see Figure 1). The wafer stage is divided into a measurement side and an exposure side. There are two chucks on the wafer stage, one at the measurement side for holding the wafer that is being measured and the other at the exposure side for holding the wafer which is being exposed. Thus, except for the very first and the last wafer, there are always two wafers being processed at the same time. When one wafer is ready to be transferred for exposure and another already exposed wafer is ready to be unloaded, a Chuck Exchange occurs. Such a Chuck Exchange requires synchronization between the measurement side and the exposure side which are otherwise independent. The processing of wafers by first measuring them accurately and then exposing them to print patterns is performed by execution of so called *sequences*. The sequence involved in the exchange of chucks is called *Chuck Exchange Sequence*. The *Chuck Exchange Sequence* is a throughput critical sequence in the TWINSCAN Lithography System as during the execution of *Chuck Exchange Sequence* neither of the two wafers is getting processed.

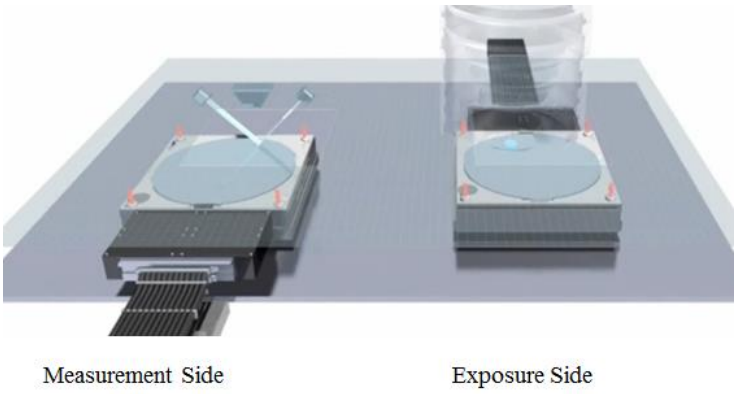


Figure 1: Wafer Stage of ASML TWINSCAN Lithography System

The execution of the sequences that are involved in exposing a wafer in an ASML TWINSCAN Lithography System is carried out by a layered Execution Framework. One of the layers within this Execution Framework is the Synchronization Control layer which controls all the resources in the system. This layer is responsible for the synchronization of sequence actions. It consists of two Synchronization Controllers, one for controlling the exposure side and the other for controlling the measurement side. These Synchronization Controllers are independent of each other. Except during Chuck Exchange, all the resources are always being synchronized by either one of the Synchronization Controller. During the execution of the *Chuck Exchange Sequence*, these independent Synchronization Controllers are involved in a two-master one slave scenario (see Figure 2) since then they are controlling a common set of resources. This introduces a potential race condition. This project investigates this potential race condition and proposes solutions to make the system robust.

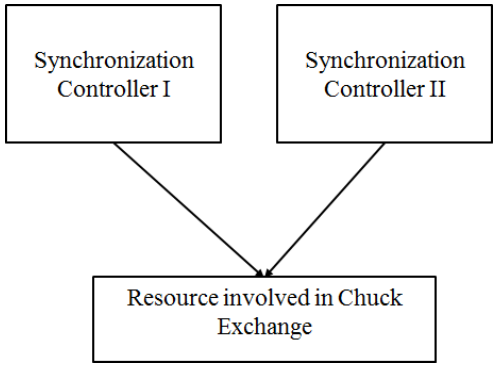


Figure 2: Two Master One Slave Scenario

1.3 Approach

In order to investigate the potential race condition, a model based approach is used. First of all, the potential race condition is analyzed in a detailed executable model of the Execution Framework. This Execution Framework Model is expressed in POOSL (Parallel Object Oriented Specification Language). Refer [4].

By reverse engineering the implementation of the *Chuck Exchange Sequence* by the Execution Framework, a Functional Specification of the Chuck Exchange, refraining from the implementation details, is created. An Implementation Specification of the *Chuck Exchange Sequence* is available which is used to create the Functional Specification. The Functional Specification describes all the resources involved in *Chuck Exchange Sequence* together with the actions that are mapped on these resources and their functional dependencies.

The Functional Specification is used to get insight in the sequence and identify the location in the *Chuck Exchange Sequence* at which the race condition might occur. An analysis reveals that the root cause is a particular dependency between two actions that are mapped on a common resource in the Functional Specification. Based on this insight an Implementation Rule is formulated which avoids the potential occurrence of race conditions. To obtain a robust solution the existing Implementation Specification is adapted by enforcing this Implementation Rule. It is shown that the robust Implementation Specification still respects the Functional Specification. Moreover, it is shown that the original Implementation Specification and the robust Implementation Specification result in equivalent Gantt charts, which implies that the robust solution leaves the performance intact.

This modified Implementation Specification is then implemented in the detailed POOSL model of the Execution Framework and is analyzed to increase the evidence of the correctness of the solution.

This report presents the industrial case study performed to investigate the robustness of the Chuck Exchange in an abstract way. The details about the execution of Chuck Exchange and the actual *Chuck Exchange Sequence* that is executed are not mentioned in this report. The working of the Execution Framework of the system is explained only briefly leaving out the ASML specific terminology and protocols. For the details about the Execution Framework and the *Chuck Exchange Sequence* we refer to the Preparation Report of this Project [1].

1.4 Report Structure

This report is further organized as follows.

Chapter 2 describes the essential concepts required to express the Functional Specification of the *Chuck Exchange Sequence*. Chapter 3 analyses the potential race condition. To this end, the Execution Framework is discussed together with the essential concepts to express the Implementation Specification of the *Chuck Exchange Sequence*. The potential race condition is analyzed and its root cause is identified and explained at the level of the Implementation Specification. Based on these insights we formulate an Implementation Rule to avoid race conditions altogether. In Chapter 4 we work towards a robust solution. An improved Implementation Specification is designed in a stepwise fashion and the rationale behind each design step is given. The improved Implementation Specification enforces the Implementation Rule and avoids the potential race condition. The robust Implementation Specification is shown to respect the Functional Specification. In addition it is shown that the original and improved Implementation Specifications yield the same timing performance. The Implementation Specification is then implemented in the executable model and analyzed in detail to underpin the correctness of the solution. Finally, Chapter 5 provides a conclusion about the work performed together with directions for future research.

Chapter 2

This chapter defines essential concepts to explain the Functional Specification of the *Chuck Exchange Sequence*. The Functional Specification for the *Chuck Exchange Sequence* itself that is created is not mentioned in this chapter because of reasons of confidentiality. This Functional Specification is designed by reverse engineering the existing Implementation Specification (see Chapter 4) of the *Chuck Exchange Sequence*. The actions and the resources involved in the *Chuck Exchange Sequence* are defined in this chapter together with the mapping of actions on the resources and the dependencies between the actions. At the end of this chapter a snapshot of the Gantt chart of the actual Functional Specification is presented.

2.1 Single Action and Multi Action

There are two types of actions involved in Chuck Exchange. These are Single Actions and Multi Actions. In the Execution Framework, these Single Actions and Multi Actions are issued by either one of the two Synchronization Controllers (see Chapter 3) to the resource (or resources). A resource executes an action. A Single Action is an action which is mapped on a single resource. A Multi Action is an action which is mapped on one or more than one resource. Therefore, the execution of a Multi Action requires synchronization of all the resources involved. All the resources involved in a Multi Action are synchronized by either one of the two Synchronization Controllers that has issued the Multi Action to the resources. The protocol involved in the synchronization of a Multi Action is not relevant from the point of view of the case study that is explained in this report. The main point of focus is that the Synchronization Controller is involved in the execution of a Multi Action on the addressed resources. Unlike, a Multi Action, a Single Action does not require synchronization.

2.2 Resources involved in Chuck Exchange

The resources involved in Chuck Exchange are responsible for the execution of Single and Multiple Actions. Every resource can execute only one action at a time. While executing a Multi Action, the resources involved should be synchronized in time by the Synchronization Controller that has issued the Multi Action at the resources.

2.3 Action-to-Resource Mapping

In order to map all the actions involved in Chuck Exchange on the resources, it is essential to study the dependencies between the actions. These dependencies are known as Functional Dependencies which should be respected in order to have the expected execution of the *Chuck Exchange Sequence*.

Since a resource can execute only one action at a time, every action that is executed on a resource has a functional dependency on the next action that is executed on that resource.

The mapping of action to resources with or without functional dependencies between the actions is explained in the following section. All the possible mapping scenarios are covered.

2.3.1 Any two actions mapped with at least one common resource

1. If two Single Actions are mapped on the same resource, then they always have either an indirect or a direct dependency between them. See Figure 3. This implies all actions mapped on a resource are executed sequentially.

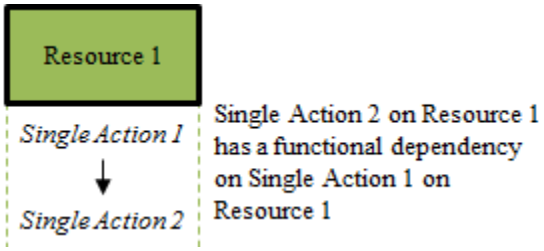


Figure 3: Functional Dependency between two Single Actions mapped on same resource

2. If a Single Action and a Multi Action share a common resource, then they always have either an indirect or a direct dependency between them. See Figures 4 and 5.

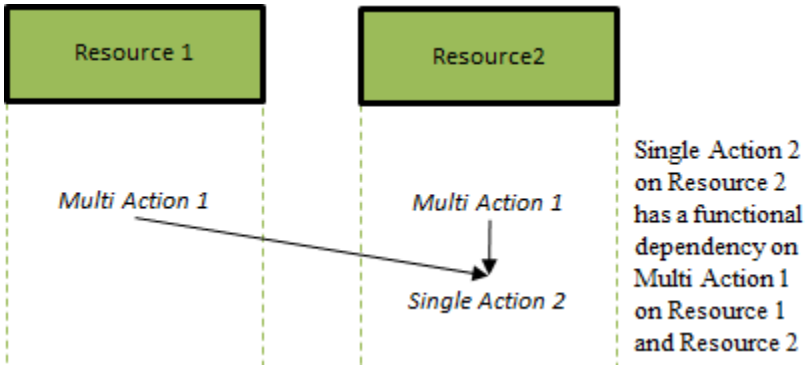


Figure 4: Functional Dependency between a Single Action and a Multi Action mapped on a common resource

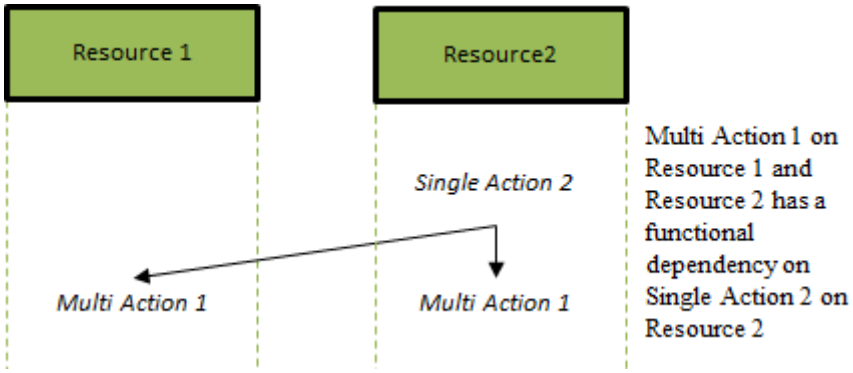


Figure 5: Functional Dependency between Multi Action and Single Action on a common resource

3. If two Multi Actions share at least one resource, then they always have either an indirect or a direct dependency between them. See Figure 6.

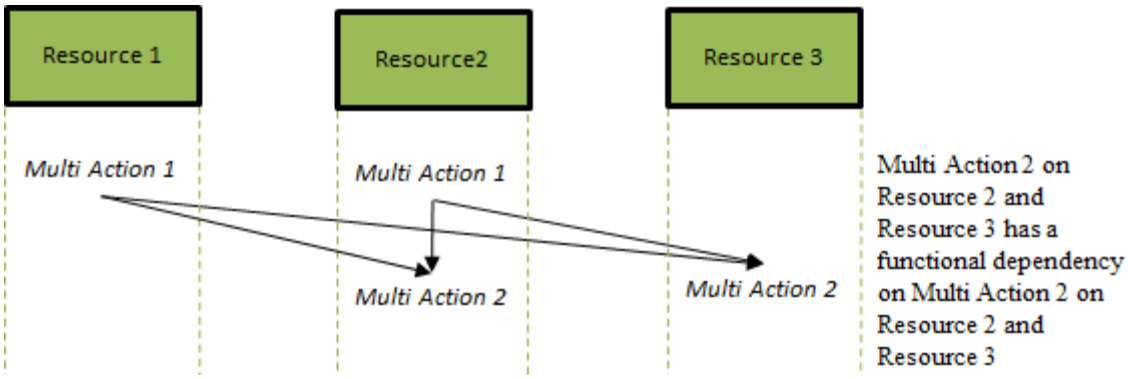


Figure 6: Functional Dependency between two Multi Actions mapped on resources with at least one common resource

2.3.2 Any two actions with no resources in common

4. If two Single Actions do not have a common resource, then either they have a dependency between them (see Figure 7) or they are executed independently (see Figure 8).

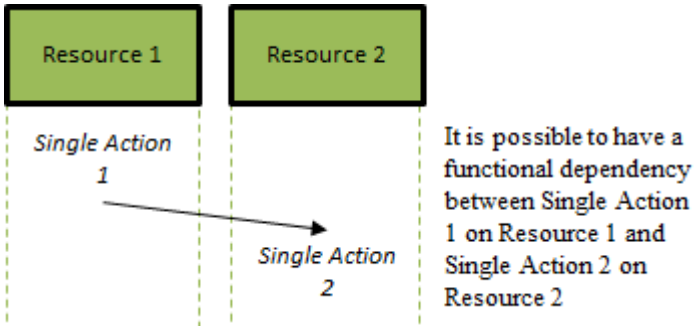


Figure 7: Functional Dependency between two Single Actions mapped on different resources

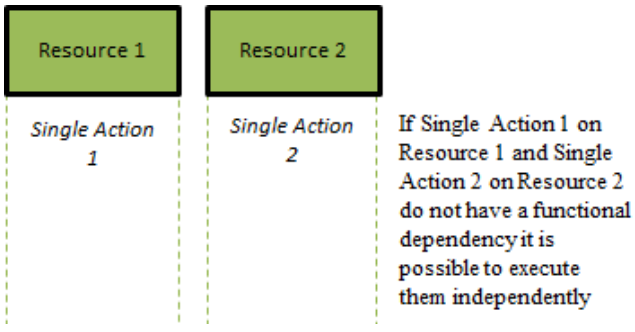


Figure 8: Independent execution of two Single Actions on different resources

5. If a Single Action and a Multi Action do not have a common resource, then either they have a dependency between them (see Figures 9 and 10) or they are executed independently (see Figure 11).

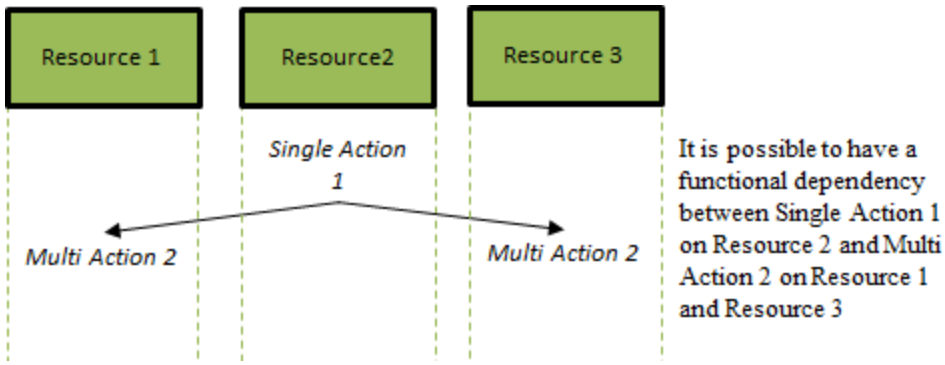


Figure 9: Functional Dependency between a Single Action and a Multi Action mapped on different resources

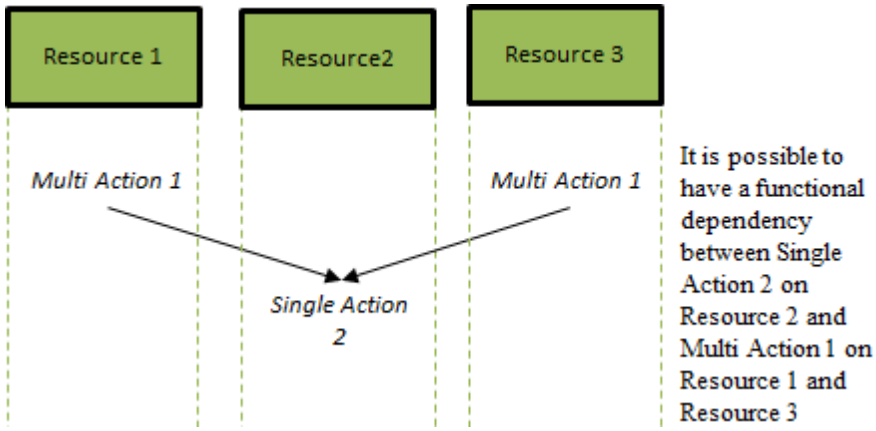


Figure 10: Functional Dependency between a Multi Action and a Single Action mapped on different resources

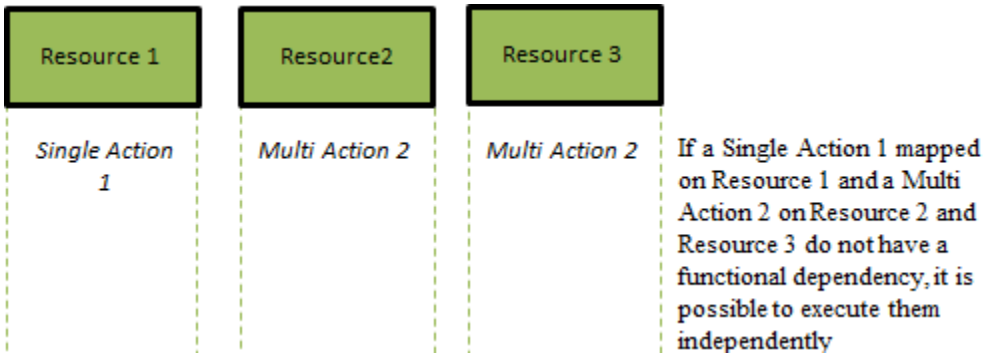


Figure 11: Independent execution of a Single Action and a Multi Action mapped on different resources

6. If two Multi Actions do not have a common resource, then either they have a dependency between them (see Figure 12) or they are executed independently (see Figure 13).

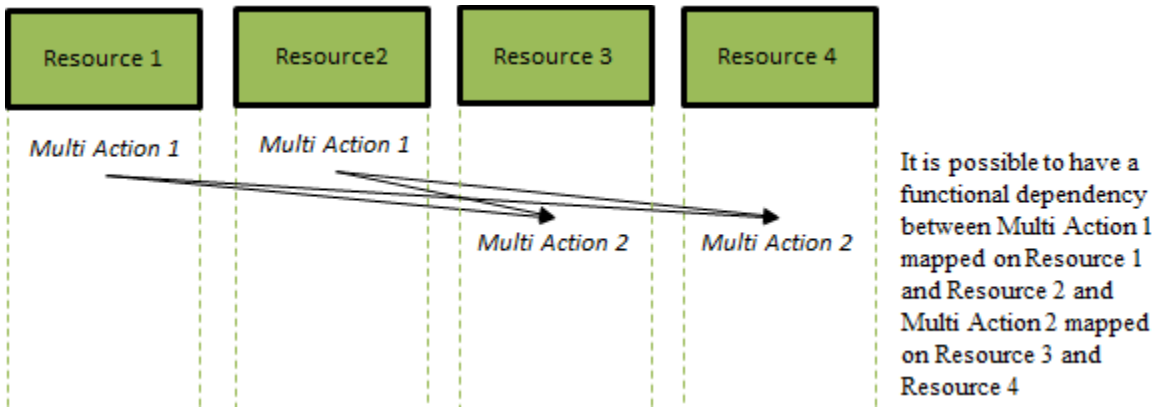


Figure 12: Functional Dependency between two Multi Actions mapped on different resources

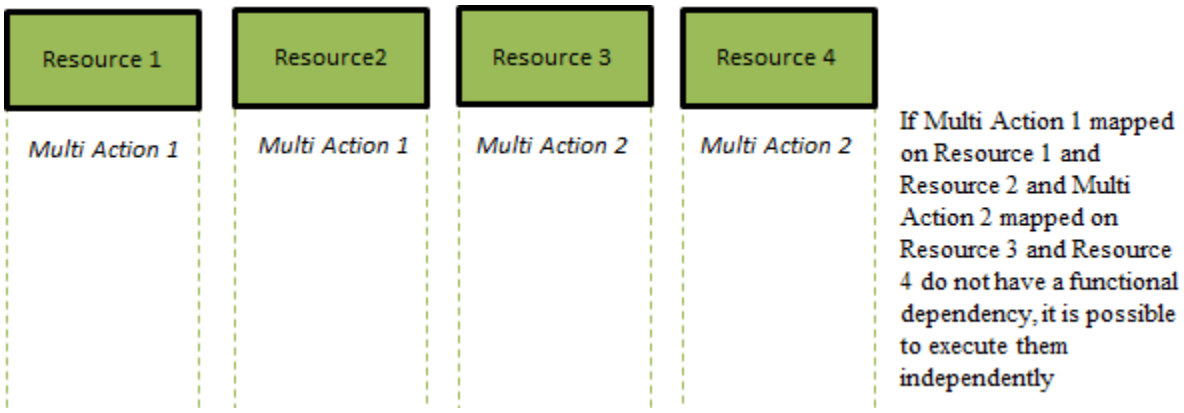


Figure 13: Independent execution of two Multi Actions mapped on mutually exclusive resources

2.4 Functional Specification for Chuck Exchange

Based on the six rules specified above a Functional Specification for the *Chuck Exchange Sequence* was created. This Functional Specification induces a Directed Acyclic Graph (DAG) consisting of all the actions (which include Single Actions and Multi Actions) and the functional dependencies between each of these actions that are involved in Chuck Exchange. This Functional Specification is verified with the domain experts and adapted accordingly.

The Functional Specification for Chuck Exchange is expressed in the Wafer Handler Logistic Tool developed by TNO for ASML [5]. This tool also generates a Gantt chart which shows the execution of actions on the resources. Figure 14 shows a snapshot of this Gantt chart. The entire Functional Specification is included in Reference [2].

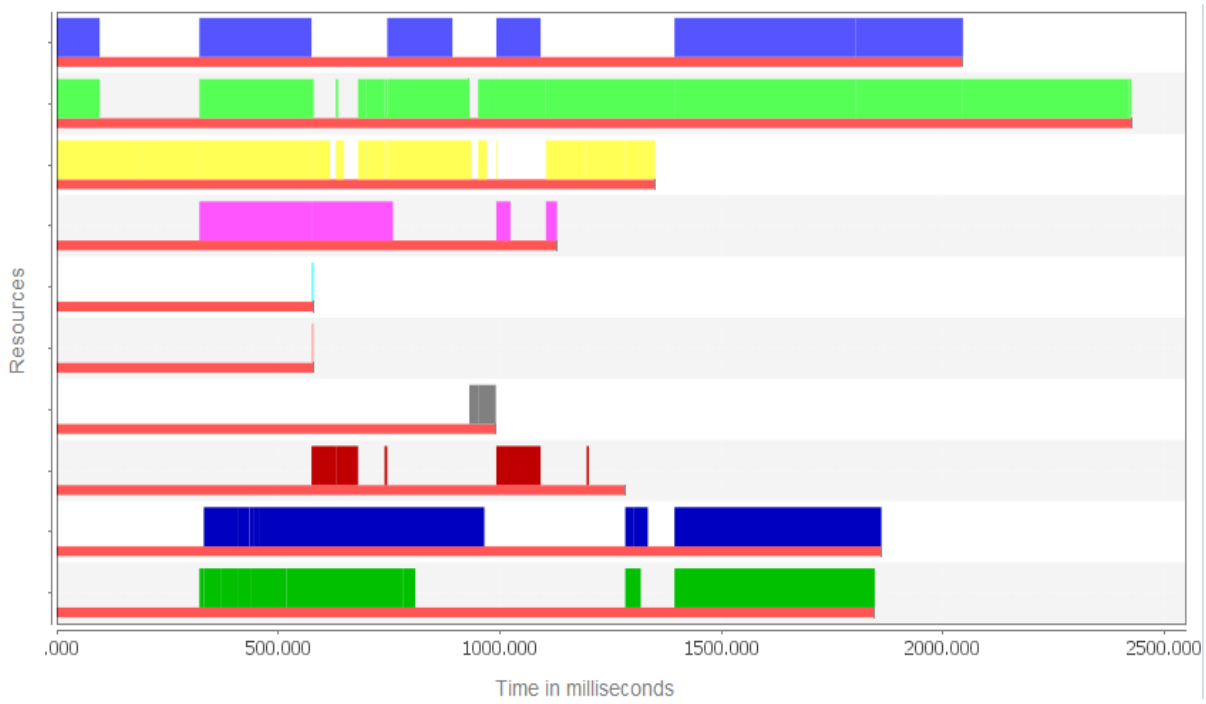


Figure 14: Gantt chart of the Chuck Exchange Functional Specification

In Figure 14 the resources involved in Chuck Exchange are shown on the Y axis and the X axis shows the actions executed on these resources in time. In the Wafer Handler Logistics Tool, resources need to be claimed before executing actions on the resources. The orange bar at every resource indicates that the resource is claimed. After all the actions mapped on a resource are executed, the resource is released.

2.5 Summary

In this chapter the concepts involved in deriving the Functional Specification for Chuck Exchange are defined. The rules for mapping actions on the resources along with their functional dependencies were formalized. These rules were applied to generate the Functional Specification for Chuck Exchange. In the next chapter, the potential race condition is analyzed in detail. To this end, the concepts to express the Implementation Specification of the *Chuck Exchange Sequence* are elaborated together with the Execution Framework that implements this specification. The analysis of the potential race condition is performed in a detailed executable model. In addition the root cause of this possibility is identified and explained at the level of the Implementation Specification.

Chapter 3

This chapter analyzes the potential race condition in detail. First the Execution Framework is explained in Section 3.1. Then the essential concepts to express an Implementation Specification are elaborated in Section 3.2 and 3.3. In Section 3.4 the possible race condition is analyzed on the level of the Execution Framework and the Implementation Specification of the *Chuck Exchange Sequence.* An Implementation Rule that always avoids the potential race conditions is formalized. The detailed executable model that is used in order to analyze the potential race condition is also briefly explained in this section.

3.1 Execution Framework

The Execution Framework that is responsible for the execution of sequences is a layered framework consisting of four layers. The components in each layer which are responsible for the execution of *Chuck Exchange Sequence* are shown in the Figure 15 along with their interfaces.

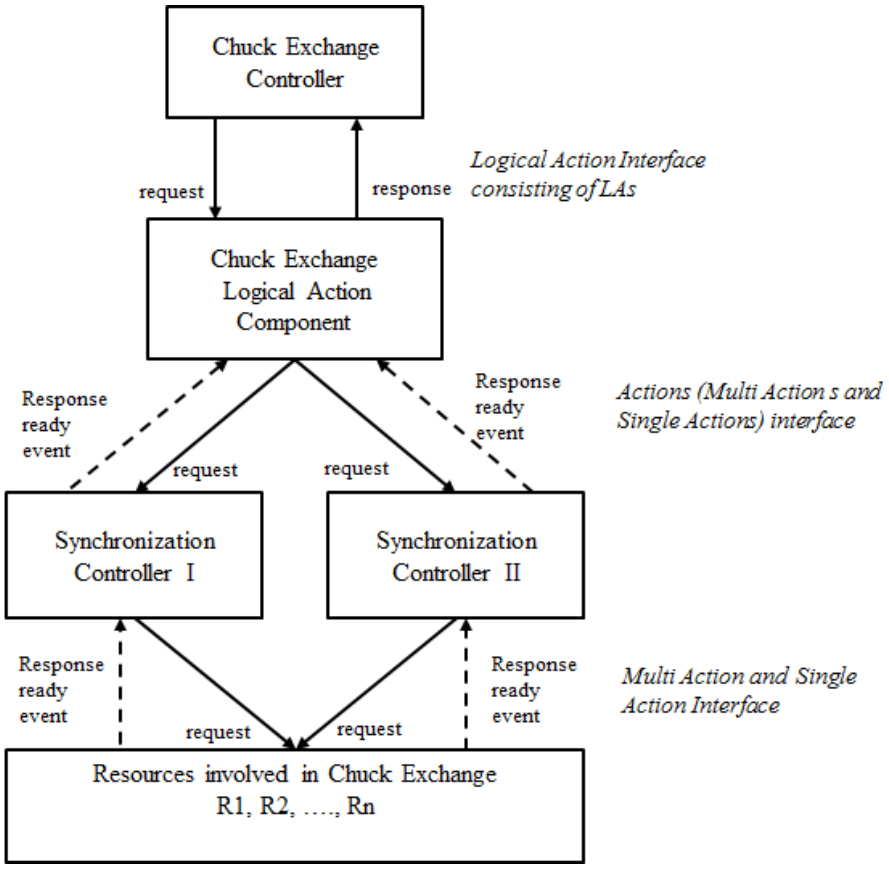


Figure 15: Execution Framework of Chuck Exchange

The four layers of the Execution Framework are the Chuck Exchange Controller at the top and below that the Chuck Exchange Logical Action Component. Then there is the Synchronization

Controller layer which consists of two Synchronization Controllers (I and II). The Resource layer at the bottom contains all the resources that are involved in the Chuck Exchange. Each layer in the Execution Framework and the interfaces between the layers is discussed in an abstract way below. A detailed explanation is found in [1].

3.1.1 Chuck Exchange Controller

This layer executes a Directed Acyclic Graph (abbreviated as DAG) of so-called Logical Actions (see Figure 16). Every node in the DAG is a Logical Action (abbreviated as *LA*). The Logical Actions are issued sequentially to the Chuck Exchange Logical Action Component, respecting the dependencies in the DAG. The issuing order of the Logical Actions for the DAG shown in Figure 16 is as follows.

LA1→LA2→LA3→LA4→LA5→LA6→LA7→LA8

Two types of dependencies exist. A Start-Start dependency implies that the target Logical Action (e.g. LA3) is issued immediately after issuing the source Logical Action (e.g. LA2). A Finish-Start dependency implies that the execution of the Source Logical Action (e.g. LA1) first has to be completed before the target Logical Action (e.g. LA3) is issued. To this end the Chuck Exchange Logical Action Component notifies the Chuck Exchange Controller if a Logical Action has completed.

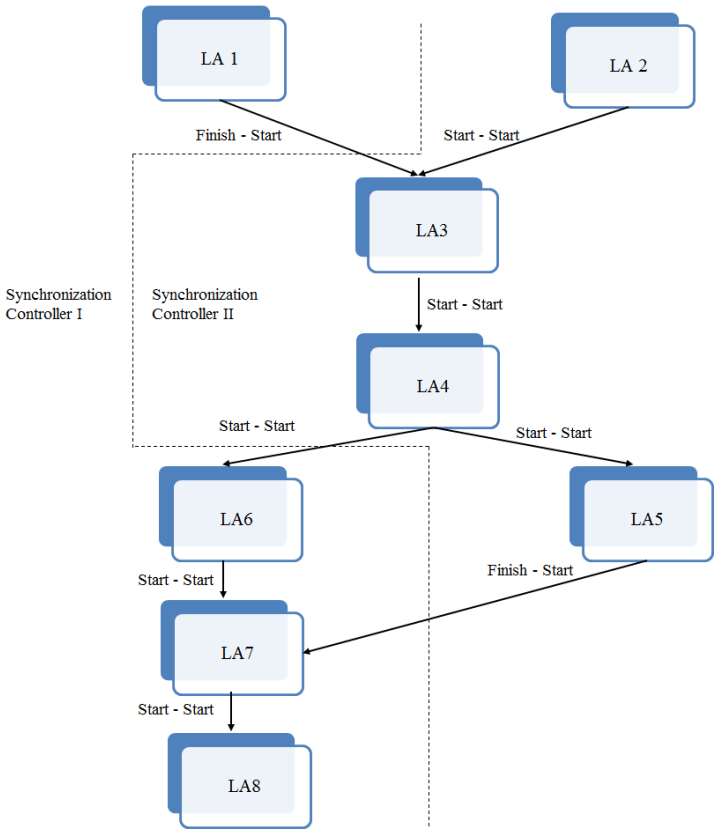


Figure 16: Chuck Exchange DAG

3.1.2 Chuck Exchange Logical Action Component

This layer receives request to execute Logical Actions that are issued by the Chuck Exchange Controller and stores them in a buffer. Every Logical Action is decomposed into one or more Actions (Single Actions or Multi Actions). An Action contains information about the functionality that is to be executed during Chuck Exchange. For example a specific X, Y, Z movement is to be carried out, then the details about the distance that is to be covered in each direction, the resources which are involved in the movement and the Synchronization Controller that is responsible for the synchronization of the movement is specified in such an Action. For each Logical Action, the Chuck Exchange Logical Action Component issues the constituent Actions in a sequential manner to the target Synchronization Controller. The target Synchronization Controllers for different Logical Actions is indicated in Figure 16. Thus, the Synchronization Controller I deals with the Actions belonging to LA1, LA6, LA7 and LA8 and Synchronization Controller II deals with LA2, LA3, LA4 and LA5. Notice that all the constituent Actions of any Logical Action are assigned always to the same Synchronization Controller.

The Chuck Exchange Logical Action Component retrieves the results of the issued Actions from the Synchronization Controller. The Synchronization Controller raises an event to notify the Chuck Exchange Logical Action Component to retrieve the results of the Action. After all the results of the Actions belonging to a Logical Action are available, the Chuck Exchange Controller can retrieve the results from the Chuck Exchange Logical Action Component.

3.1.3 Synchronization Controller layer

This layer consists of two independent Synchronization Controllers (I and II). Each Synchronization Controller receives Actions from the Chuck Exchange Logical Action Component and stores them sequentially in a local queue. Each Action is either a Single Action or a Multi Action (see Chapter 2). The Synchronization Controller issues these Single Actions and Multi Actions sequentially to the resources that are addressed by each of these actions. Even though, actions are issued sequentially, they can still be executed by the resources concurrently, which is explained in detail in Section 3.3 .

The Synchronization Controller retrieves the results of actions from the Resource layer by subscribing to an event at the Resource layer. After finishing the execution of an action, the resource that executed it raises an event to notify the Synchronization Controller to retrieve the result

3.1.4 Resource layer

This layer contains all the resources that are involved in the execution of Chuck Exchange. Resources do not communicate with each other. They receive actions (Single Actions and Multi Actions) from a Synchronization Controller and queue them in their local queue. A resource can execute only one action at a time as explained in Chapter 2. The queuing and execution of actions at a resource is governed by a State Machine. We refer to our Preparation Phase Report

[1] for details about the State Machine. An abstract version is shown in Figure 17. The state of the resource is determined by the status of its queue. This state can either be Idle, Partly Filled, Full or Empty. The queue is empty in the Idle and Empty states. In the Partly Filled state, the queue at the resource contains at least on action. The Full state indicates that the queue at the resource has reached its specified capacity of the maximal number of actions that can be queued at the resource. The transitions to the State Machine are issued by the Synchronization Controller. The transitions mainly concern the issuing of actions or retrieving of results for the issued actions. The Synchronization Controller receives an update of the state of the resource after issuing every transition and depending upon the state of the resource at the Synchronization Controller, the Controller decides when to issue the next transition to the resource.

The protocols summarized in the above section are abstracted from the original ASML protocols for the reasons of confidentiality. The abstraction is sufficient to explain the potential race condition in Section 3.4. In order to fully understand the flow of requests and response in the Execution Framework we refer to [1].

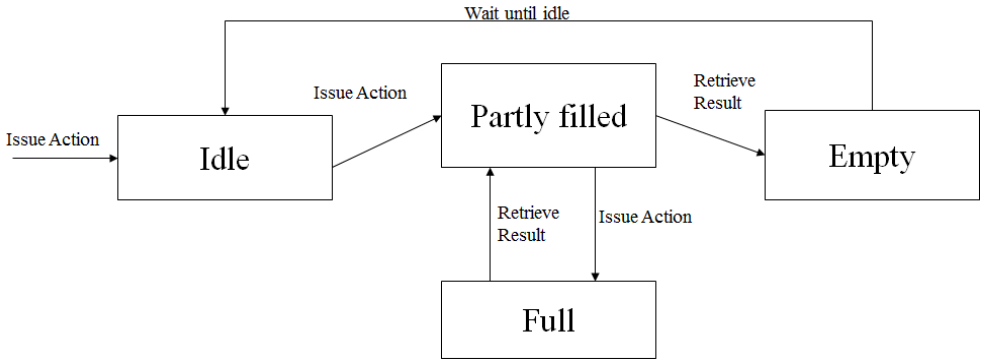


Figure 17: Resource layer State Machine

3.2 Implementation Concepts

The implementation concepts used in the Execution Framework include Passive Claims, Issuing Order, Synchronization Instances and Finish-Start & Start-Start dependencies. Each of these implementation concepts are defined below. The example for each implementation concept is mentioned in the next section where the implementation of the six rules of Action-to-Resource mapping (discussed previously in Chapter 2) in the Execution Framework is shown.

3.2.1 Passive Claims

Passive claims of resources are used to implement functional dependencies between actions on different resources. For instance if Action A on resource R1 has a functional dependency to Action B on resource R2, this dependency can be implemented by letting A put a claim on R2. Passive claims are explained in detail in Section 3.3.

3.2.2 Issuing Order

A functional dependency is automatically established between two actions that are executed on the same resource. The dependency is enforced through the issuing order of these two actions at the Synchronization Controller. For instance, if Single Action A has a functional dependency to Single Action B on the same resource, this dependency is established by first issuing action A and then action B. It is important to note that in order to establish such dependency, they should be dealt with by the same Synchronization Controller.

The Synchronization Controller is responsible of respecting dependencies between two Multi Actions. A Synchronization Controller has a Synchronization Driver which acts as one of the resources while executing a Multi Action. Thus, all Multi Actions always have one resource in common which is the Synchronization Driver. Therefore, the issuing order of Multi Actions by the Synchronization Controller always determines a functional dependency between two Multi Actions issued by same Synchronization Controller. Hence, if Multi Actions are issued by the same Synchronization Controller they always have a direct or indirect functional dependency. The details about synchronization of a Multi Action can be found in [6].

3.2.3 Synchronization Instance

There are two independent Synchronization Controllers in the Execution Framework which are responsible for the synchronization of the *Chuck Exchange Sequence*. The Logical Actions in the DAG shown in Figure 16 address a particular Synchronization Controller in order to execute the Single Actions and Multi Actions belonging to the Logical Actions.

3.2.4 Finish-Start & Start-Start dependencies

Finish-Start and Start-Start dependencies are applied at the Chuck Exchange Controller to enforce dependencies between two Logical Actions. If a Finish-Start dependency exists between two Logical Actions then the next Logical Action is issued by the Chuck Exchange Controller only after the previous Logical Action has finished its execution and the result has been retrieved by the Chuck Exchange Controller.

If a Start-Start dependency exists between two Logical Actions then the next Logical Action is issued by the Chuck Exchange Controller as soon as the previous Logical Action has been queued at the Synchronization Controller layer. Hence in case of a Start-Start dependency, the Chuck Exchange Controller does not wait for the result of the first issued Logical Action. Typically, when two Logical Actions are addressed to the same Synchronization Controller, a Start-Start dependency is used and when two Logical Actions are addressed to different Synchronization Controllers a Finish-Start dependency is used.

This synchronization concept is illustrated by the following example. Consider three Logical Actions LA5, LA6 and LA7 (see Figure 18). These Logical Actions contain Actions each of which can be either a Multi Action or a Single Action. These actions are shown in the green colored blocks inside every Logical Action in the Figure 18. The number mentioned after the

name of every Action is a two digit number separated by *dot*. The digit before the dot indicates the issuing order of the Logical Action to which the Action (Single Action or Multi Action) belongs and the digit after the dot specifies the issuing order of the Actions themselves at the corresponding Synchronization Controller. For example, LA6 contains Multi Action 6.1. In Multi Action 6.1, *'1'* indicates that the issuing order of the Multi Action 6.1 is 1 and *'6'* indicates the issuing order of the Logical Action (LA 6) in the DAG (see Figure 16) which is 6.

Now LA6 and LA7 have a Start-Start dependency, which means that LA7 can be issued immediately after LA6 is queued while LA5 and LA7 have a Finish-Start dependency, which means that LA5 has to finish its execution and send its result to the Chuck Exchange Controller after which LA7 is issued.

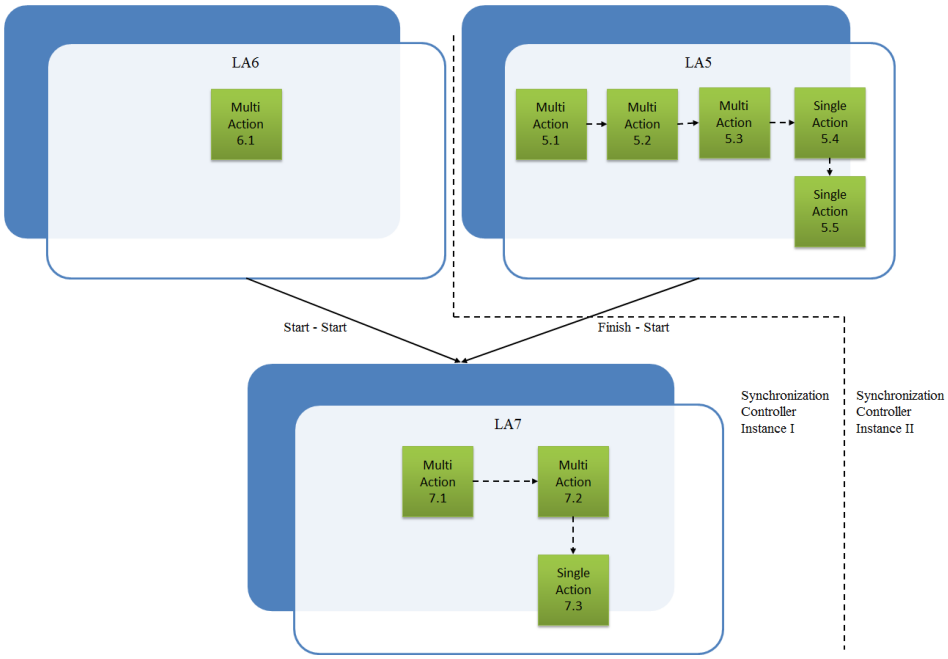


Figure 18: Example of Finish-Start & Start-Start dependencies

3.3 Action-to-Resource Mapping in the Execution Framework

This section describes how an Action-to-Resource Mapping (seen in Chapter 2) is realized in the Execution Framework. The mapping of Single Actions and Multi Actions on resources in the diagrams below shows the execution order of Single Actions and Multi Actions on resources along with their functional dependencies and the implementation concepts applied in order to respect the functional dependencies. These diagrams extent the Functional Specifications as described in Chapter 2 by annotating them with the previously explained implementation concepts (Synchronization Controller Instantiation, Passive Claims, Issuing Order and Start-Start / Finish-Start dependencies). Therefore we will call these diagrams Implementation Specifications.

The annotations are as follows.

Colors (Red and Blue) are used to differentiate between the Actions issued by the two Synchronization Controllers. The issuing order of the actions is denoted by the number in the box that precedes the name of the action. The number mentioned in the box is a two digit number separated by *dot*. The digit before the dot indicates the issuing order of the Logical Action to which the Action (Single Action or Multi Action) belongs and the digit after the dot specifies the issuing order of the Actions themselves at the corresponding Synchronization Controller. In the diagrams below, a grey color is used to indicate a Start-Start or Finish-Start dependency between Logical Actions. The Single Actions and Multi Actions belonging to different Logical Actions are grouped separately and the Start-Start or Finish-Start dependency between the Logical Actions is specified by a grey colored arrow (with either a Start-Start or Finish-Start label) between the two groups of actions. A Passive Claim at a resource is indicated by a dotted box enclosing the action and the resource that has been claimed passively by the action.

In the next subsection we will explain the alternative implementation choices to implement functional dependencies. For this, we will follow the same structure as in Section 2.3 in which the alternative functional dependencies are explained.

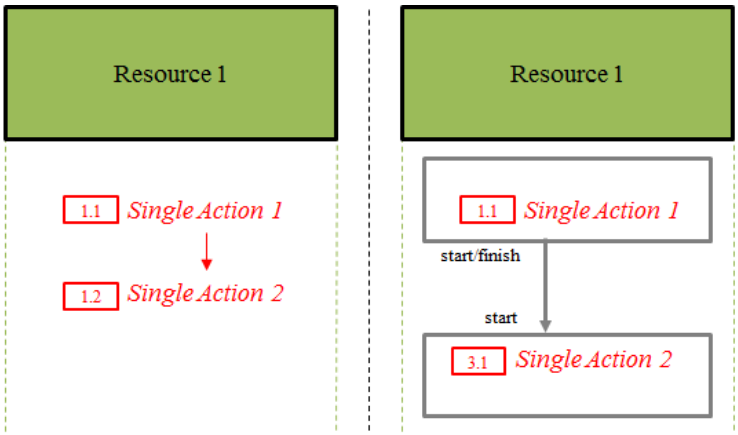
3.3.1 Any two actions mapped on resources with at least one common resource:

1. If two Single Actions are mapped on the same resource, then they always have either an indirect or a direct dependency between them.

   Two ways exists to implement rule 1 in the Execution Framework.

   a. If the two Single Actions are issued by the same Synchronization Controller then, the functional dependency between them is implemented by either the issuing order of the two Actions if the two Actions belong to the same Logical Action, or by the issuing order of the Logical Actions to which they belong. In the case where, the two Actions belong to different Logical Actions, the functional dependency between them can be implemented by either having a Finish-Start or Start-Start dependency between the two Logical Actions. See Figure 19.

   b. If the Synchronization Controller issuing the two actions are different, then the dependency is implemented a Finish-Start dependency enforced between two Logical Actions to which the two Actions belong. It is important to note here that implementation of a Finish-Start dependency results in a performance penalty for all the Actions in the target Logical Action. This performance penalty results because of the fact that the implementation of a Finish-Start dependency between two Logical Actions introduces additional functional dependencies between the Actions corresponding to the two Logical Actions which are in fact not required. See Figure 20.



Single Action 1 and Single Action 2 belong to same Logical Action and the issuing order of the two actions implement the functional dependency between them

Single Action 1 and Single Action 2 belong to different Logical Actions which are addressed to same Synchronization Controller instance and the issuing order of the Logical Actions implement the functional dependency between them.

Figure 19: Implementation of Functional dependency between two Single Actions issued by the same Synchronization Controller in the Execution Framework

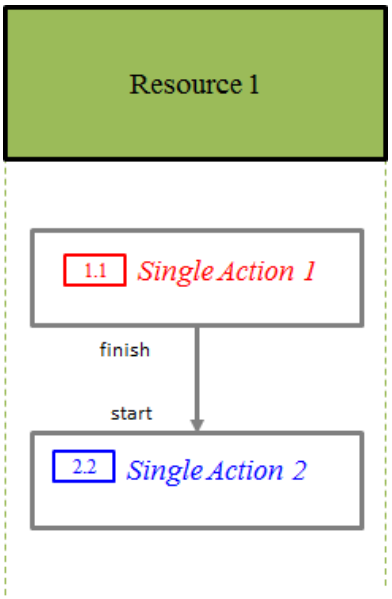


Figure 20: Implementation of Functional dependency between two Single Actions issued by different Synchronization Controllers in the Execution Framework

2. If a Single Action and a Multi Action share a common resource, then they always have either an indirect or a direct dependency between them.
   Two ways exists to implement rule 2 in the Execution Framework.

a. If the Single Action and Multi action are issued by the same Synchronization Controller then, the dependency between them is implemented by either the issuing order of the two actions if the two actions belong to the same Logical Action, or by the issuing order of the Logical Actions to which they. In the case where, the two Actions belong to different Logical Actions, the functional dependency between them can be implemented by either having a Finish-Start or Start-Start dependency between the two Logical Actions. See Figures 21 and 22.
b. If the Synchronization Controllers issuing the two Actions are different then the dependency is implemented by a Finish-Start dependency enforced between the Logical Actions to which the Single Action and Multi Action belong. It is important to note here that implementation of a Finish-Start dependency results in a performance penalty for all the Actions in the target Logical Action. This performance penalty results because of the fact that the implementation of a Finish-Start dependency between two Logical Actions introduces additional functional dependencies between the Actions corresponding to the two Logical Actions which are in fact not required. See Figure 23.

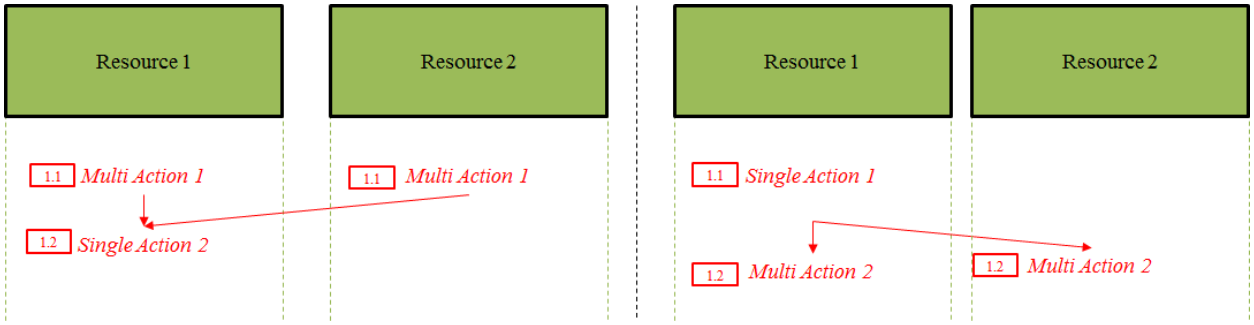


Figure 21: Implementation of a functional dependency between a Single Action and a Multi Action belonging to the same Logical Action mapped on a common resource and issued by the same Synchronization Controller in the Execution Framework. (Figure 21 shows the two possible combinations of dependencies between a Multi Action and a Single Action)

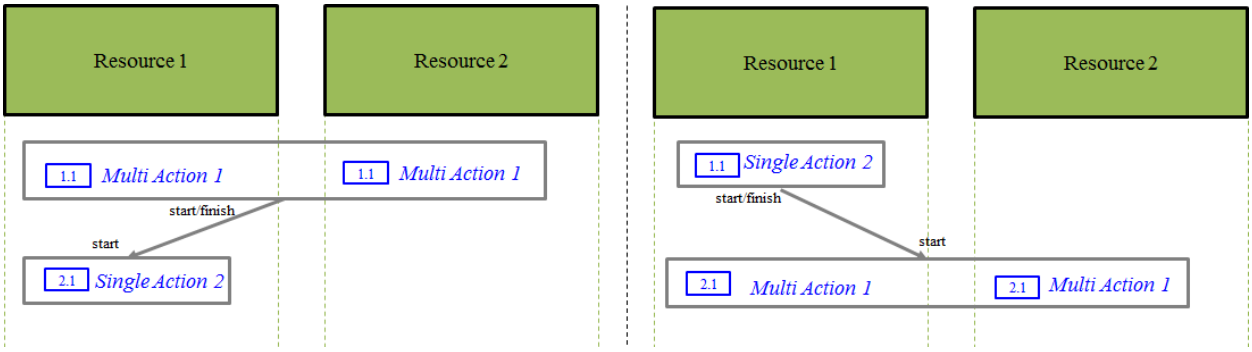


Figure 22: Implementation of a functional dependency between a Single Action and a Multi Action belonging to different Logical Actions mapped on a common resource and issued by the same Synchronization Controller in the Execution Framework. (Figure 22 shows two possible combinations of dependencies between a Multi Action and a Single Action)

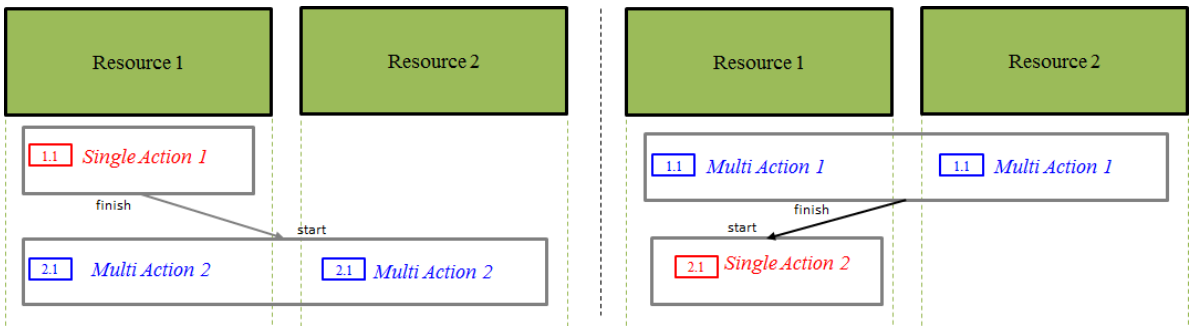


Figure 23: Implementation of a functional dependency between a Single Action and a Multi Action belonging to different Logical Actions mapped on a common resource and issued by different Synchronization Controllers in the Execution Framework. (Figure 23 shows two possible combinations of dependencies between a Multi Action and a Single Action)

3. If two Multi Actions share at least one resource, then they always have either an indirect or a direct dependency between them.

   Two ways exists to implement rule 3 in the Execution Framework.

   a. If the two Multi Actions are issued by the same Synchronization Controller then, the dependency between them is respected by either the issuing order of the two actions if the two actions belong to the same Logical Action, or by the issuing order of the Logical Actions to which they belong. In the case where, the two Actions belong to different Logical Actions, the functional dependency between them can be implemented by either having a Finish-Start or Start-Start dependency between the two Logical Actions. See Figure 24.

   b. If the Synchronization Controllers issuing the two Actions are different then the dependency is implemented by a Finish-Start dependency enforced between the Logical Actions to which the Single Action and Multi Action belong. It is important to note here that implementation of a Finish-Start dependency results in a performance penalty for all the Actions in the target Logical Action. This performance penalty results because of the fact that the implementation of a Finish-Start dependency between two Logical Actions introduces additional functional dependencies between the Actions corresponding to the two Logical Actions which are in fact not required. See Figure 25.

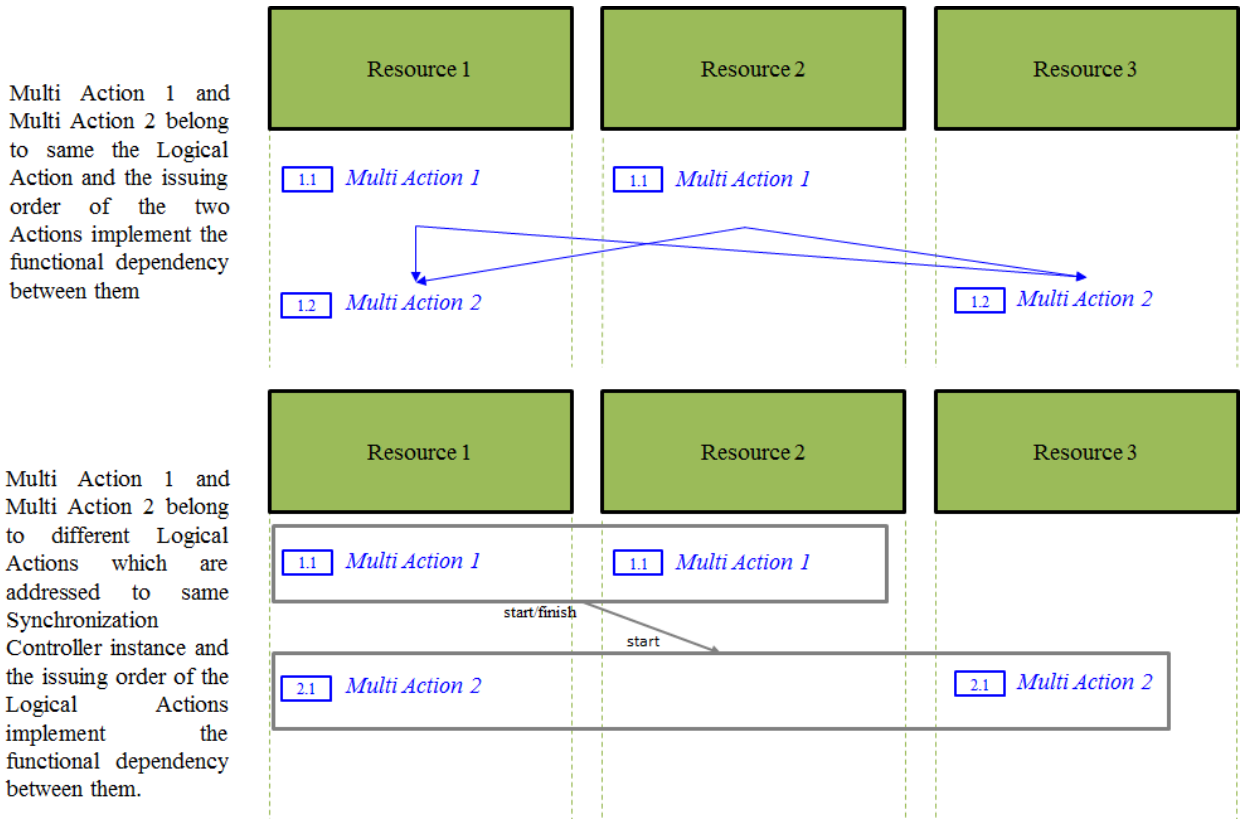


Figure 24: Implementation of functional dependency between two Multi Actions, issued by the same Synchronization Controller mapped on resources with at least one common resource in the Execution Framework (The top and the bottom cases illustrate two possible scenarios in which the dependencies between two Multi Actions can be realized)

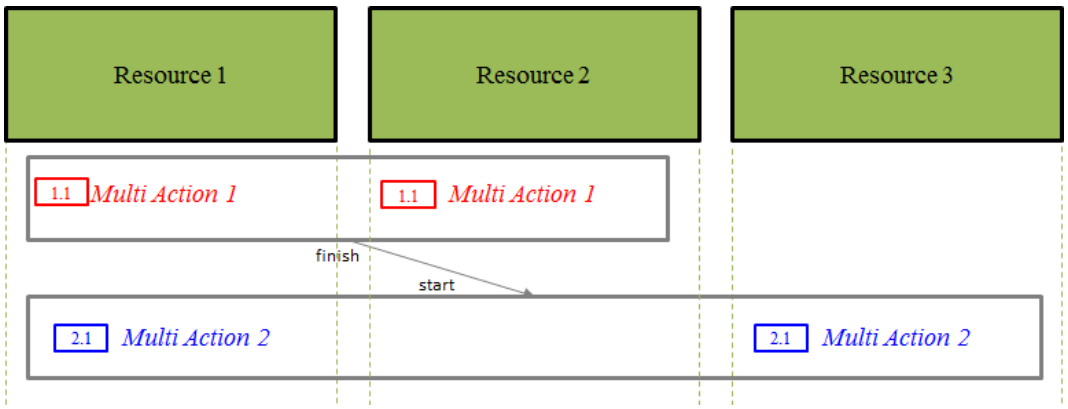


Figure 25: Implementation of functional dependency between two Multi Actions issued by different Synchronization Controllers, mapped on resources with at least one common resource in the Execution Framework

3.3.2 Any two actions with no resources in common

4. If two Single Actions do not have a common resource, then either they have a dependency between them or they are executed independently.
   Three ways exists to implement rule 4 in the Execution Framework.
   a. If two Single Actions do not have a common resource and have a dependency between them, then this dependency is implemented by a Passive Claim. It is important that the two Single Actions are issued by the same Synchronization Controller. The Synchronization Controller makes sure that no Actions are issued to the passively claimed resource until the Action that claimed the resource passively has finished its execution. See Figure 26. It is important to note that if the two Actions belong to different Logical Actions that are addressed to the same Synchronization Controller then the issuing order of the two Logical Actions along with the Passive claim implement the functional dependency. Here a Start-Start dependency is implemented between the two Logical Actions. If a Finish-Start dependency is implemented between two Logical Actions then a Passive Claim is not required to implement the functional dependency between the two Actions. This case is shown in Figure 27.

   b. If the Synchronization Controllers issuing the two Actions are different then the dependency is implemented by a Finish-Start dependency enforced between the Logical Actions to which the Single Action and Multi Action belong. It is important to note here that implementation of a Finish-Start dependency results in a performance penalty for all the Actions in the target Logical Action. This performance penalty results because of the fact that the implementation of a Finish-Start dependency between two Logical Actions introduces additional functional dependencies between the Actions corresponding to the two Logical Actions which are in fact not required. See Figure 28.

   c. If two Single Actions do not have a common resource and they do not have a dependency between them, they are executed independently irrespective of the Synchronization Controller that issues the Actions. See Figure 29.

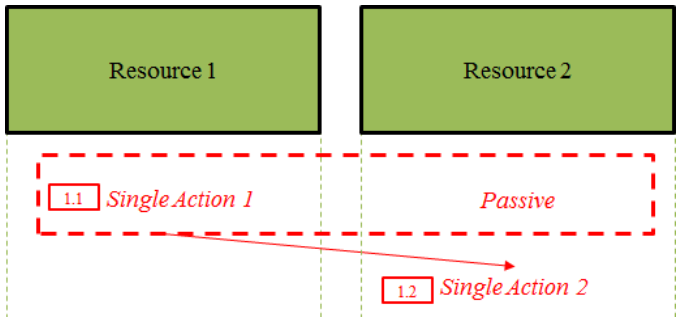


Figure 26: Implementation of functional dependency between two Single Actions belonging to the same Logical Action and mapped on different resources in the Execution Framework

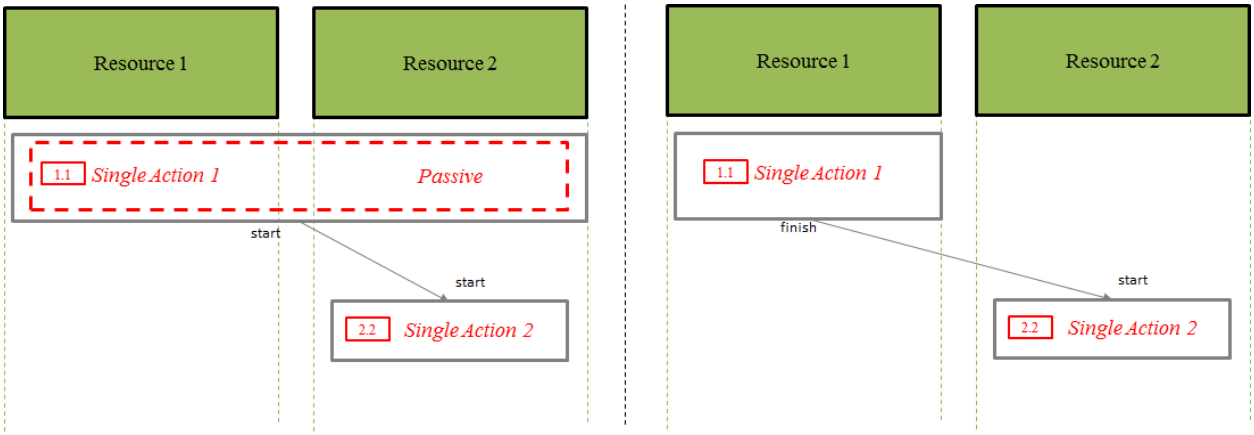


Figure 27: Implementation of functional dependency between two Single Actions belonging to different Logical Actions that are issued by the same Synchronization Controller and mapped on different resources in the Execution Framework

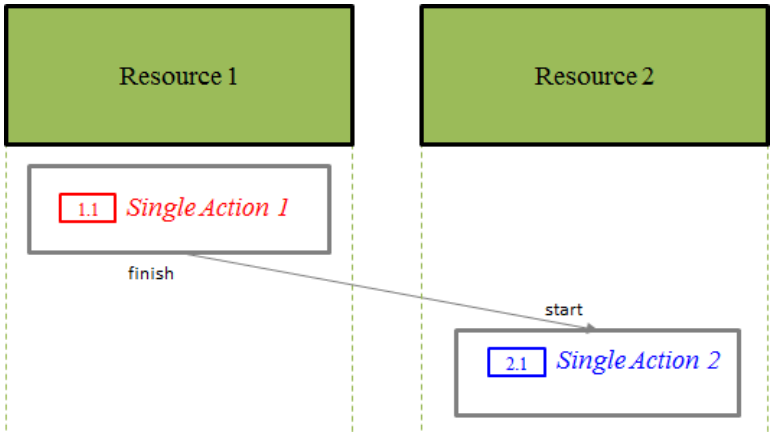
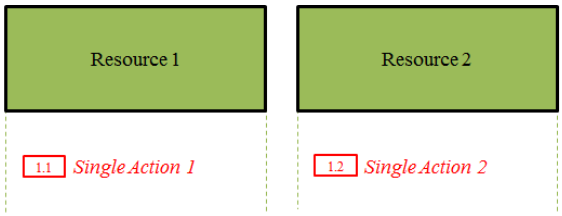
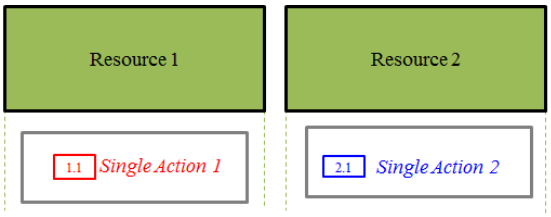


Figure 28: Implementation of functional dependency between two Single Actions issued by different Synchronization Controller and mapped on different resources in the Execution Framework



Single Action 1 and Single Action 2 are issued by same Synchronization Controller instantiation to Resource 1 and Resource 2 respectively. Although the issuing of the two Single Actions from Synchronization Controller instantiation to the resources is sequential, their execution at their respective resources is independent. The issuing of Logical Actions to which the Actions belong is also sequential.



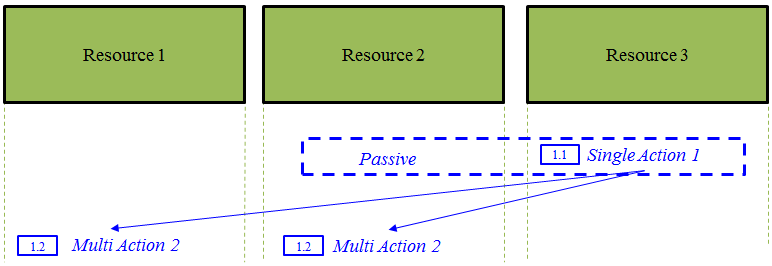
Single Action 1 and Single Action 2 are issued by different Synchronization Controller instantiation to Resource 1 and Resource 2 respectively. The issuing of the two Single Actions from the respective Synchronization Controller instantiations to the resources as well as their execution at their respective resources is independent. The issuing of Logical Actions to which the Actions belong is although sequential.

Figure 29: Independent execution of two Single Actions mapped on different resources (The figure illustrate two possible scenarios in which the dependencies between two Single Actions can be realized)

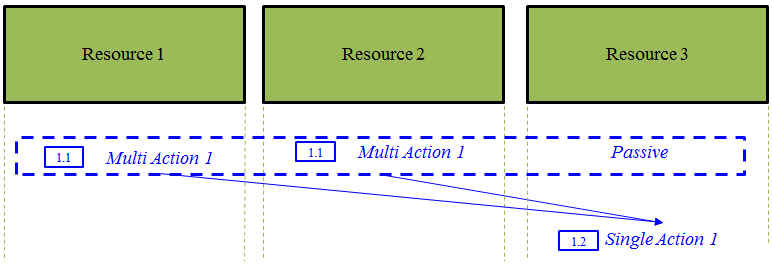
5. If a Single Action and a Multi Action do not have a common resource, then either they have a dependency between them or they are executed independently.
   Three ways exists to implement rule 4 in the Execution Framework. The explanations are similar to those in rule 4.
   a. If a Single Action and a Multi Action do not have a common resource and have a dependency between them then this dependency can be implemented by a Passive Claim. It is important that the two Actions are issued by the same Synchronization Controller. The Synchronization Controller makes sure that no Actions are issued to the passively claimed resource until the Action that claimed the resource passively has finished its execution. See Figures 30 and 31. It is important to note that if the two Actions belong to different Logical Actions that are addressed to the same Synchronization Controller, the issuing order of the two Logical Actions along with the Passive claim implement the functional dependency. Here a Start-Start dependency is implemented between the two Logical Actions. If a Finish-Start dependency is implemented between two Logical Actions then a Passive Claim is not required to implement the functional dependency between the two Actions.

   b. If the Synchronization Controllers issuing the two Actions are different then the dependency is implemented by a Finish-Start dependency enforced between the Logical Actions to which the Single Action and Multi Action belong. It is important to note here that implementation of a Finish-Start dependency results in a performance penalty for all the Actions in the target Logical Action. This performance penalty results because of the fact that the implementation of a Finish-Start dependency between two Logical Actions introduces additional functional dependencies between the Actions corresponding to the two Logical Actions which are in fact not required. See Figure 32.

   c. If a Single Action and a Multi Action do not have a common resource and they do not have a dependency between them, they are executed independently irrespective of the Synchronization Controller that issues the Actions. See Figure 33.

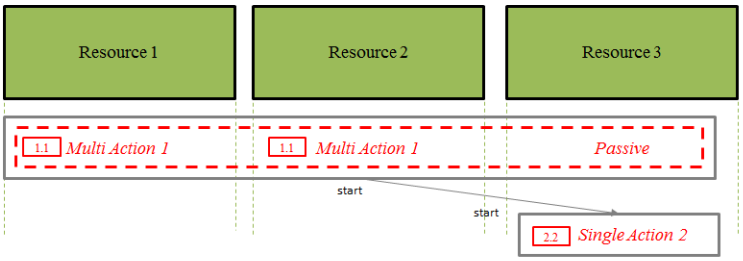


Single Action 1 passively claims one of the resources that is involved in Multi Action 2 in order to implement the functional dependency. Both actions belong to a same Logical Action.

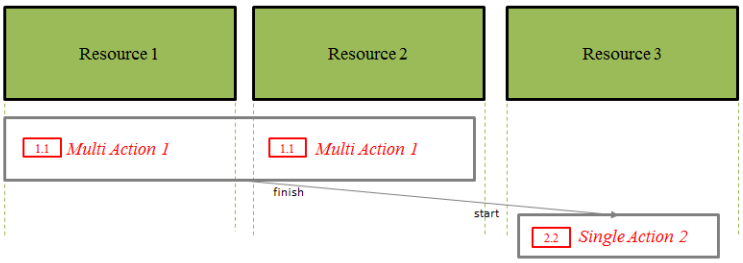


Multi Action 1 passively claims the resources that executes Single Action 2 in order to implement the functional dependency. Both actions belong to a same Logical Action.

Figure 30: Implementation of functional dependency between a Multi Action and a Single Action belonging to the same Logical Action and mapped on different resources (The top and the bottom figures illustrate the two possible combinations of dependencies between a Multi Action and a Single Action)



Multi Action 1claims passive the resource on which Single Action 2 is to be executed in order to implement the functional dependency. Both actions belong to a different Logical Action but are addressed to same Synchronization Controller instance. Thus issuing order of Logical Actions along with the Passive Claim implement the dependency between the two actions.



Multi Action 1 and Single Action 2 belong to different Logical Actions with a Finish-Start dependency between them. In this case the Finish-Start dependency between the Logical Actions implement the functional dependency between the two Actions.

Figure 31: Implementation of functional dependency between a Single Action and a Multi Action belonging to different Logical Actions and mapped on different resources

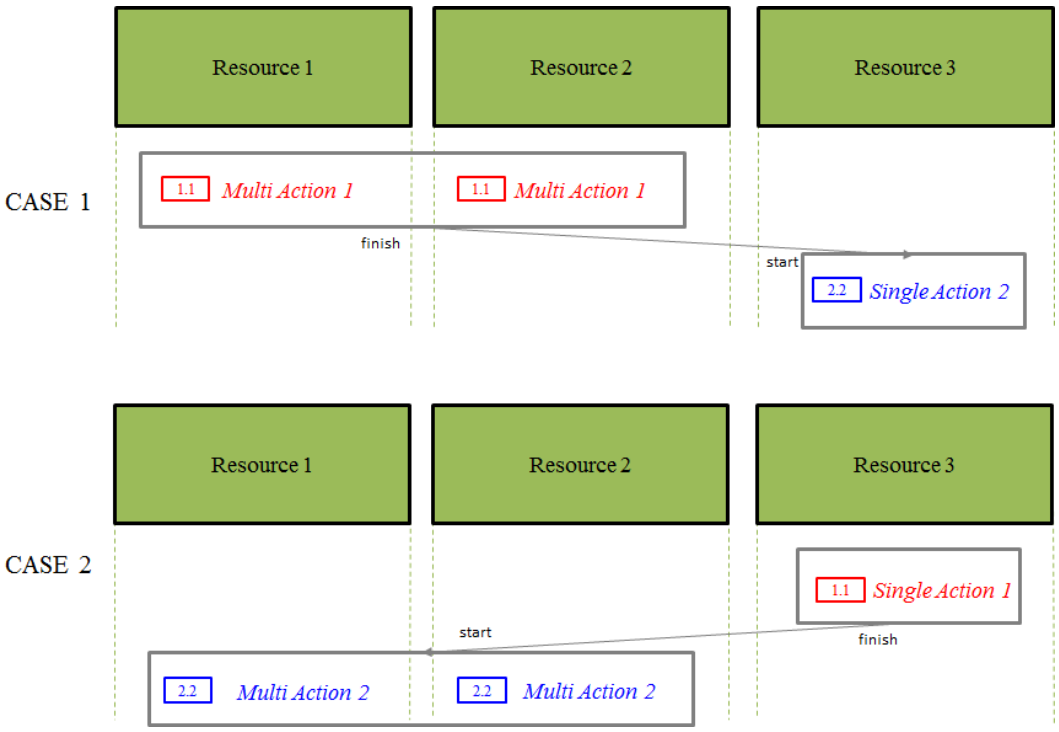
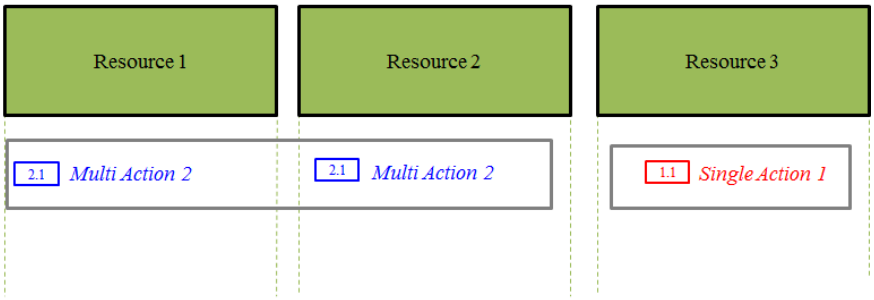


Figure 32 : Implementation of functional dependency between a Multi Action and a Single Action that are mapped on different resources and issued by different Synchronization Controllers (CASE 1 and CASE 2 show two possible combinations of dependencies between a Multi Action and a Single Action)

Single Action 1 and Multi Action 2 are issued by different Synchronization Controller instantiation to Resource 3 and Resource 1 & 2 respectively. The issuing of the two Single Actions from the respective Synchronization Controller instantiations to the resources as well as their execution at their respective resources is independent. The issuing of Logical Actions to which the Actions belong is although sequential.



Single Action 1 and Multi Action 2 are issued by same Synchronization Controller instantiation to Resource 3 and Resource 1 & 2 respectively. Although the issuing of the two Actions from Synchronization Controller instantiation to the resources is sequential, their execution at their respective resources is independent. The issuing of Logical Actions to which the Actions belong is also sequential.

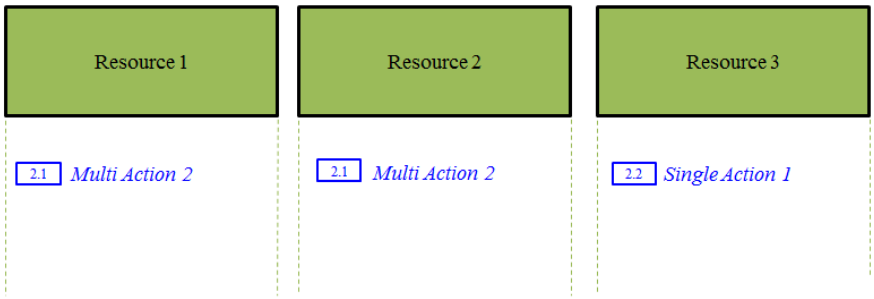


Figure 33: Independent execution of a Single Action and a Multi Action (The top and the bottom figures illustrate the two possible combinations of dependencies between a Multi Action and a Single Action)

6. If two Multi Actions do not have a common resource, then either they have a dependency between them (see Figure 12) or they are executed independently (see Figure 13).
   Three ways exists to implement rule 6 in the Execution Framework.
   a. If the two Multi Actions are issued by the same Synchronization Controller then the functional dependency between them is implemented by the issuing order of the two Multi Actions. As explained above, the Synchronization Controller has a Synchronization Driver which acts as one of the resources when executing a Multi Action. Thus, all Multi Actions always have one resource in common which is the Synchronization Driver. Therefore, the issuing order of Multi Actions by the Synchronization Controller implements the dependency between the two Multi Actions. See Figure 34. It is important to note that if the two actions belong to different Logical Actions that are addressed to the same Synchronization Controller then the issuing order of the two Logical Actions implements the functional dependency between the two Actions. In this case where, the two Actions belong to different Logical Actions, the functional dependency between them can be implemented by either having a Finish-Start or Start-Start dependency between the two Logical Actions. See Figure 35.

   b. If the Synchronization Controllers issuing the two Multi Actions are different then the dependency is implemented by a Finish-Start dependency enforced between the corresponding Logical Actions. It is important to note here that in this case the two Multi Actions are part of different Logical Actions as they are issued by different Synchronization Controllers to the resources. Implementation of a Finish-Start dependency results in a performance penalty for all the Actions in the target Logical Action. This performance penalty results because of the fact that the implementation of a Finish-Start dependency between two Logical Actions introduces additional functional dependencies between the Actions corresponding to the two Logical Actions which are in fact not required. See Figure 36.

   c. If two Multi Actions do not have a common resource and they do not have a dependency between them, they are executed independently irrespective of the Synchronization Controller that issues the Actions. See Figure 37.

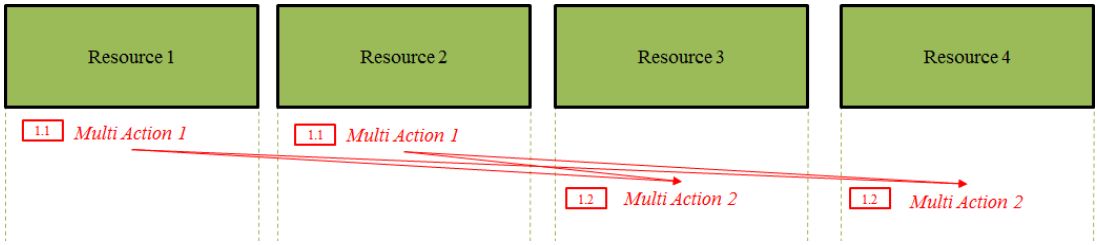


Figure 34: Implementation of functional dependency between two Multi Actions belonging to same Logical Action, mapped on different resources with no common resource

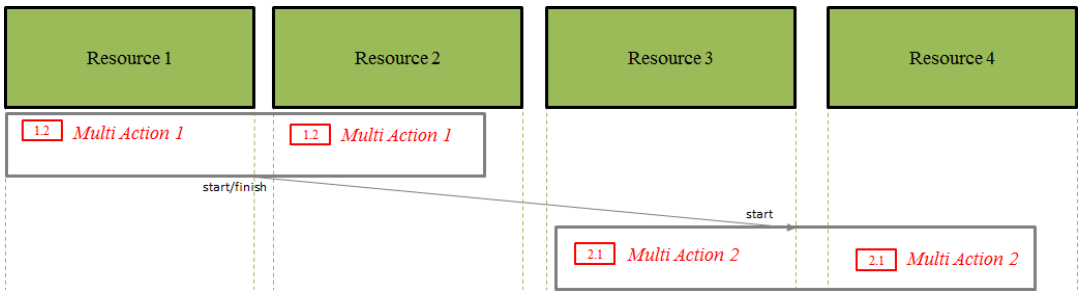


Figure 35: Implementation of functional dependency between two Multi Actions belonging to different Logical Actions, mapped on different resources with no common resource

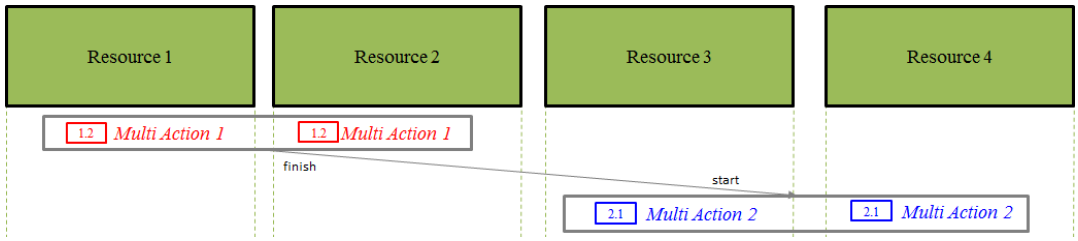
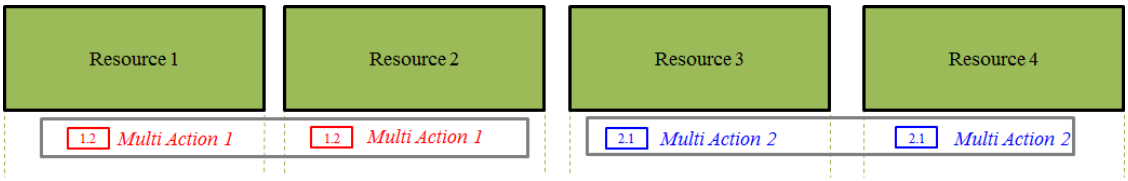


Figure 36: Implementation of functional dependency between two Multi Actions issued by different Synchronization Controllers mapped on different resources with no common resource



Multi Action 1 and Multi Action 2 belong to different Logical Actions that are addressed to different Synchronization Controller instance. Therefore, they are issued as well as executed independently. The issuing of Logical Actions to which the Actions belong is although sequential.

Figure 37: Independent execution of two Multi Actions mapped on different resources

3.4 Analysis of the potential Race Condition

In this section the root cause of the potential race condition is analyzed. The insights gained are used in the next chapter to design an implementation that satisfies the Functional Specification (explained in Chapter 2) and is robust (i.e. free of race conditions).

3.4.1 Potential occurrence of Race Condition in Implementation Specification

The part of the Functional Specification as well as the Implementation Specification which might introduce a race condition is shown in Figures 38 and 39 respectively. The annotations used in

these figures are the same as those explained in Section 3.3. The Implementation Specification of the Functional Specification shown in Figure 38 is explained below.

- Multi Action1, Multi Action 2, Multi Action 3, Single Action 4 and Single Action5 belong to LA5 of the DAG shown in Figure 16. Multi Action 6 belongs to LA6 and Multi Action 7, Multi Action 8 and Single Action 9 belong to LA7 of the DAG. The issuing order of Multi and Single Actions and the issuing order of their corresponding Logical Actions are mentioned in the box preceding the name of each Action in the Figure 39.
- LA6 and LA7 are issued by Synchronization Controller I while LA5 is issued by Synchronization Controller II.
- The functional dependency between Multi Action 1 and Multi Action 2, Multi Action 2 and Multi Action3 and between Single Action 4 and Single Action 5 is implemented through the issuing order of these Actions by Synchronization Controller II.
- Similarly, the functional dependency between Multi Action 7 & Multi Action 8 is implemented by the issuing order of these Actions by Synchronization Controller I.
- The Start-Start dependency between LA6 and LA7 implements the functional dependency between Multi Action 6 and Multi Action 7.
- The functional dependency between Multi Action 2 and Single Action 4 is implemented by a Passive Claim on Resource 3 by Multi Action 2.
- The functional dependency between Single Action 5 and Single Action 9 and Multi Action 3 and Multi Action 7 is implemented by enforcing a Finish-Start dependency between the Logical Actions LA5 and LA7. It is important to note here that the implementation of Finish-Start dependency between LA5 and LA7 introduces an additional functional dependency between Single Action 5 and Multi Action 7. Thus the Functional Specification that is derived from the original Implementation Specification for a part of the *Chuck Exchange Sequence* is different from the original Functional Specification. The Functional Specification derived from the original Implementation Specification is shown in Figure 40. The additional functional dependency however does not affect the performance. This is because Multi Action 7 has a functional dependency on Multi Action 3 and the execution time of Multi Action 3 is more than the execution time of Single Action 5. Thus, the additional functional dependency that is introduced between Single Action 5 and Multi Action 7 does not give performance penalty. This is can be seen from the Gantt charts of the Functional Specification and the original Implementation Specification that are shown in Figures 41 and 42 respectively. The numbers in the schedule denote the serial numbers of the Actions that are in the Functional Specification. For example the execution of Multi Action 1 is indicated by digit 1 in the schedule. The arrows indicate the functional dependencies between the Actions mapped on different resources. Both the Gantt charts are same except that there is an additional dependency in Figure 42 between Multi Action 7 mapped on Resource 1 and Single Action 5 mapped on Resource 3.

The Finish-Start dependency that is implemented between LA 5 and LA 7 in order to implement the functional dependency between Single Action 5 and Single Action 9 is the location where the race condition might be introduced. The reason for this is explained in the next section.

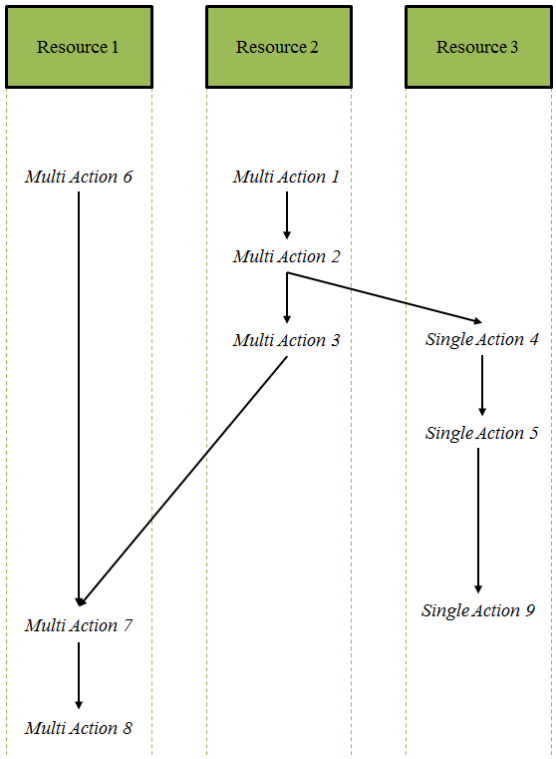


Figure 38: Functional Specification of a part of *Chuck Exchange Sequence*

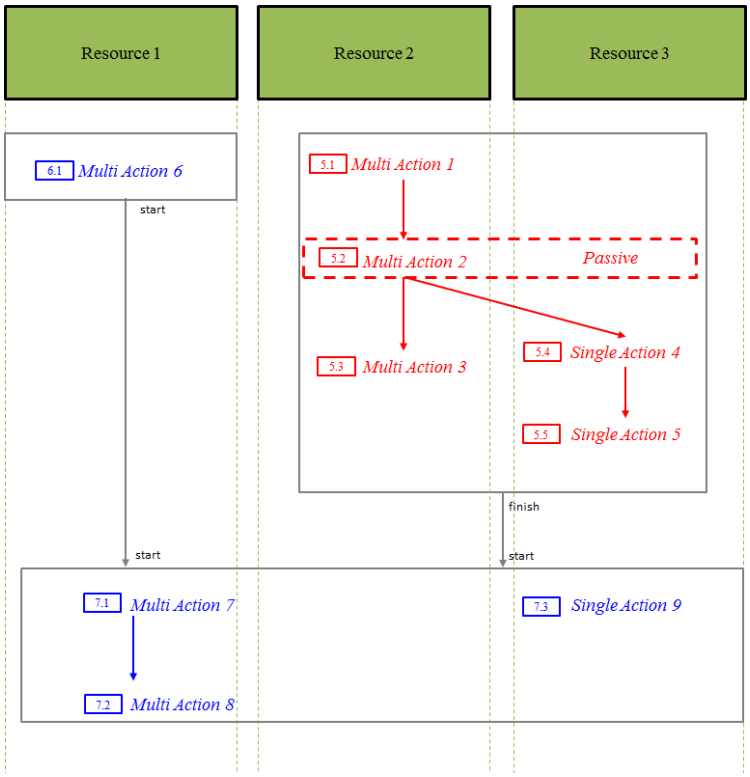


Figure 39: Implementation Specification of the Function Specification shown in Figure 38 for a part of *Chuck Exchange Sequence*

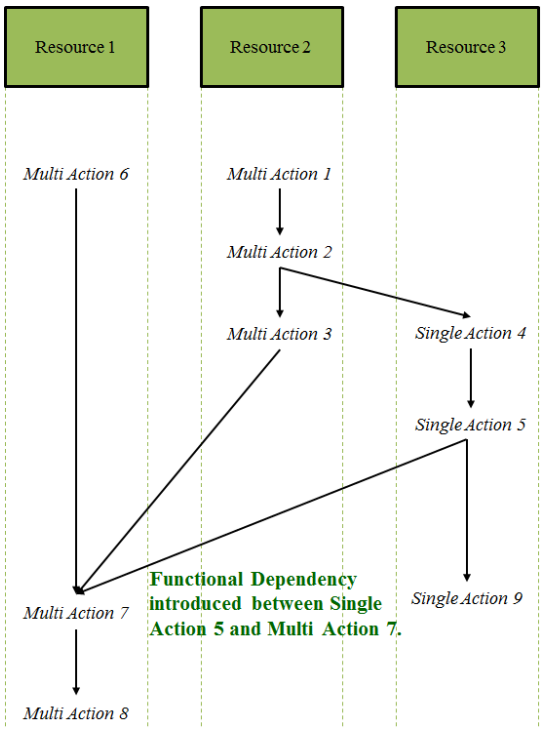


Figure 40: Functional Dependencies introduced by the original Implementation Specification

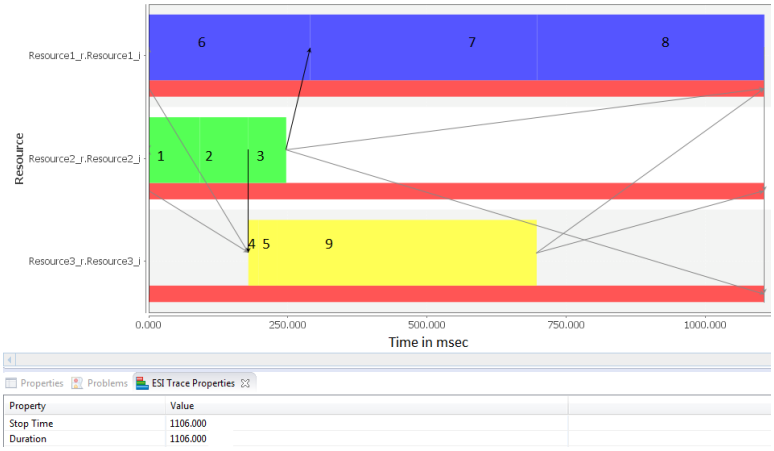


Figure 41: Gantt chart of the original Functional Specification

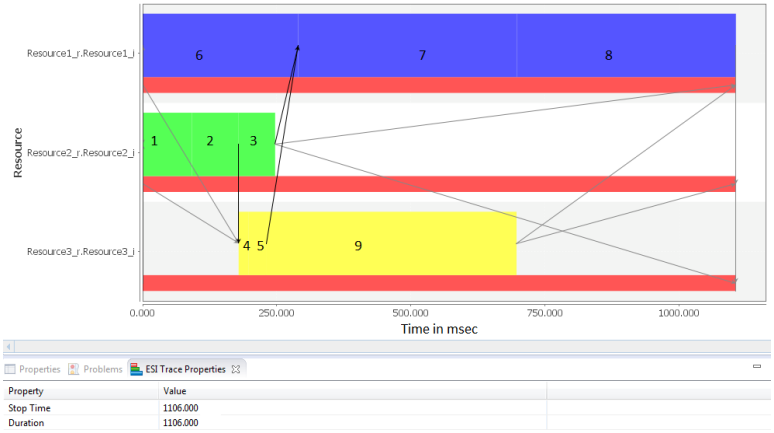


Figure 42: Gantt chart of the original Implementation Specification

3.4.2 Potential Race Condition in the Execution Framework

As explained in Section 3.1.4 the execution of actions at the resource is governed by a State Machine (see Figure 17) at each resource. The transitions in the State Machine are issued by the Synchronization Controllers. A resource can receive actions from both the Synchronization Controllers. After receiving an action the resource returns its new state to the Synchronization Controller that issued the action. If at some point in time both the Synchronization Controllers are trying to issue a transition to a resource, a race condition might occur.

Figure 43 shows the proper behavior to execute Single Action 5 and Single Action 9.The comment blocks with green colored text represent the state of the Resource 3 and the mirrored state of Resource 3 at Synchronization Controller I and Synchronization Controller II. In this case, Single Action 9 is issued after the issuing of the WAIT_UNTIL_IDLE call. Hence, the

Single Action 9 is issued when Resource 3 is in Idle State which is allowed according to the State Machine in Figure 17. However, in case the Single Action 9 is issued earlier than the WAIT_UNTIL_IDLE call, this call is received in the Partially Filled State which is not allowed. This behavior is shown in Figure 44.

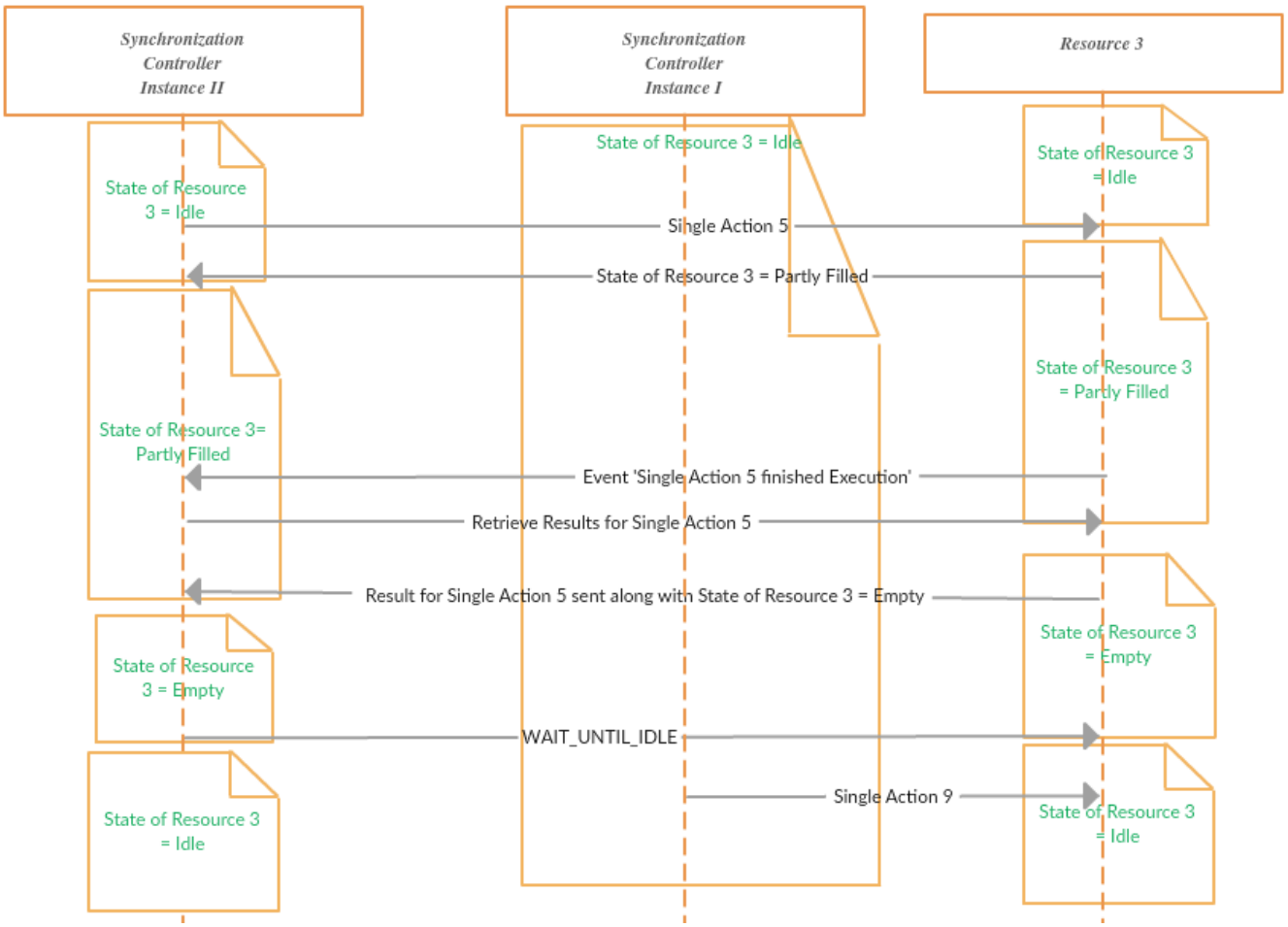


Figure 43: Proper behavior

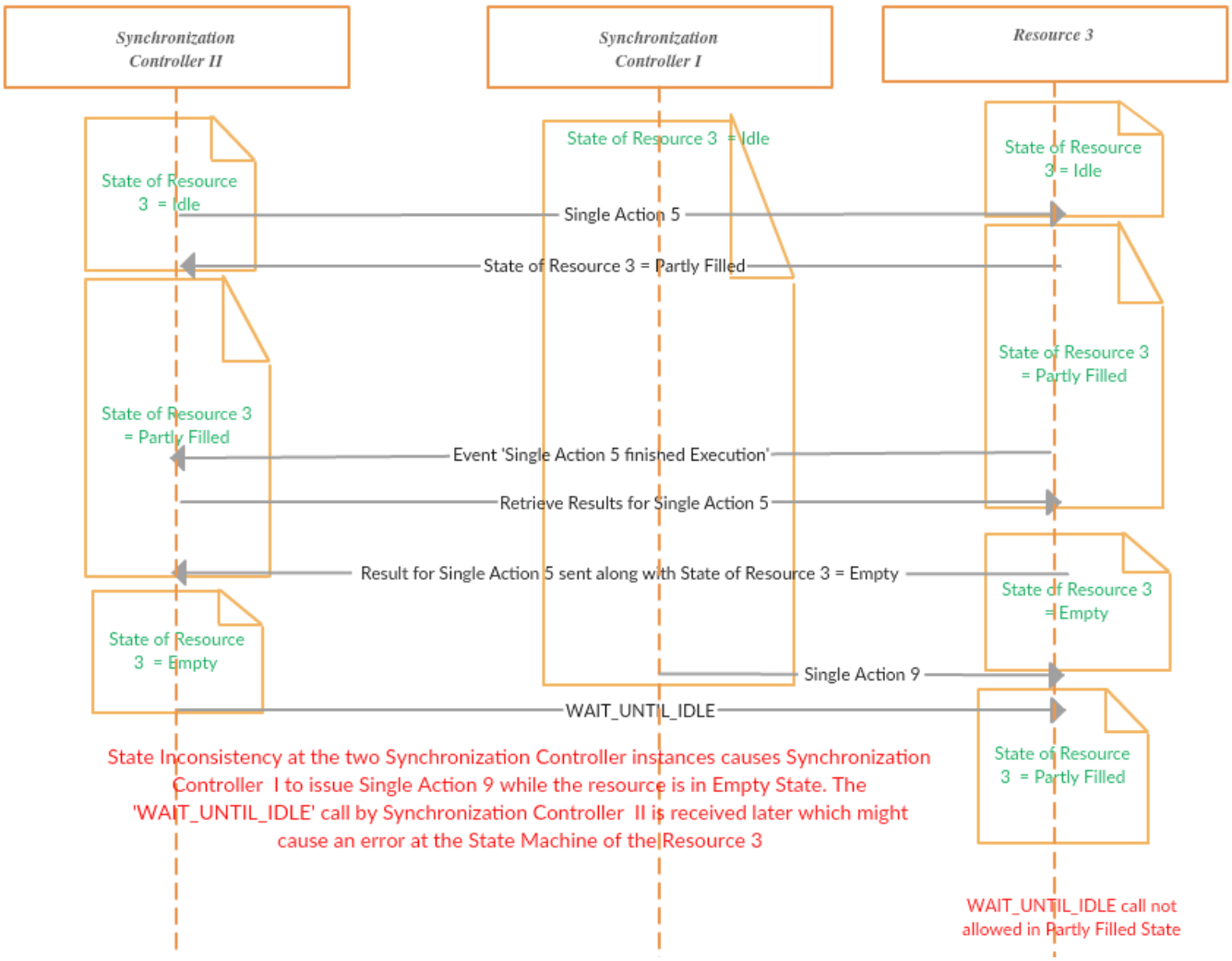


Figure 44: Possible Faulty Behavior

The race condition and faulty behavior might occur because

1. Single Action 5 and Single Action 9 are performed by the same resource (i.e. Resource 3);
2. Single Action 5 and Single Action 9 are issued by different Synchronization Controllers.

Hence, to avoid the potential race condition we should have a single Synchronization Controller dealing with both the Single Actions. This is shown in the scenario of Figure 45.

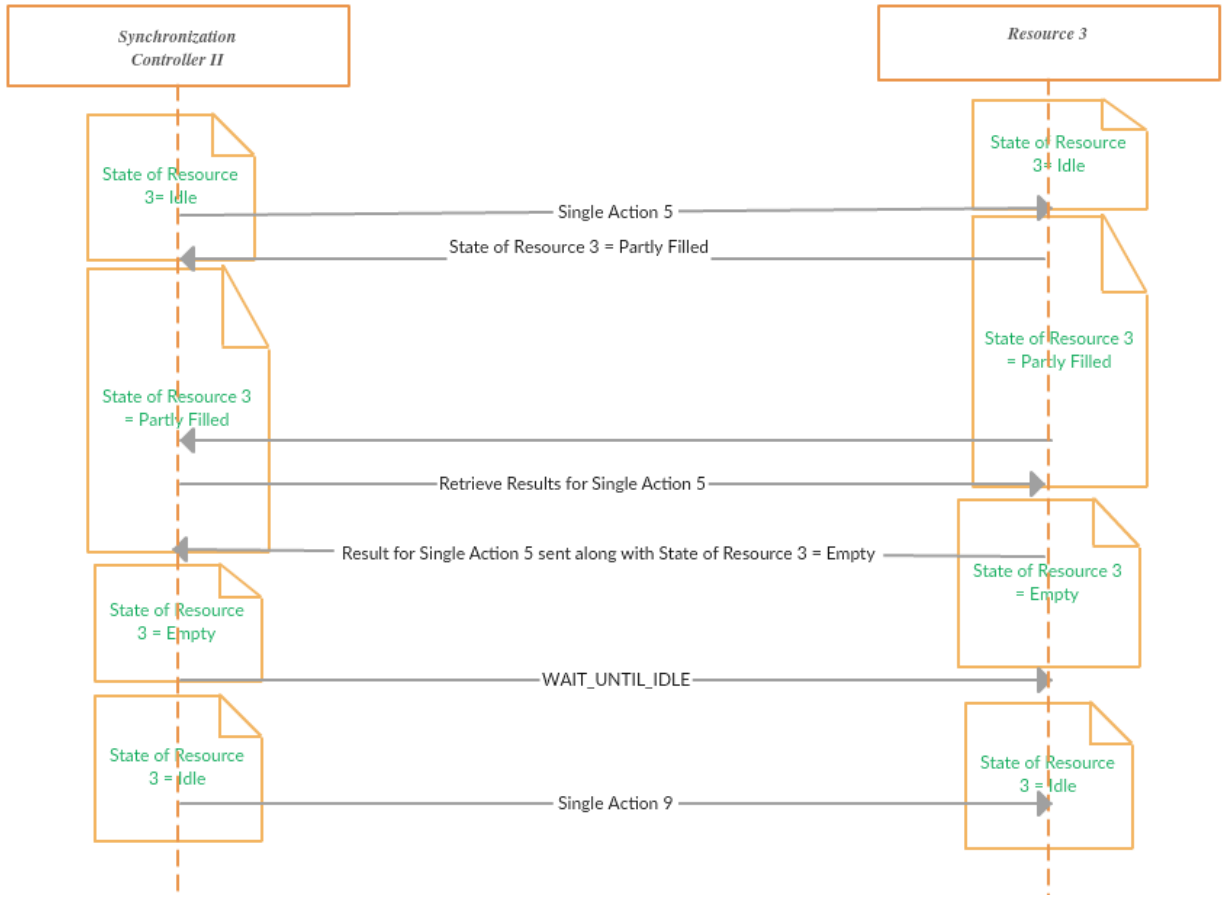


Figure 45: Potential Race condition is avoided when Single Action 5 and Single Action 9 are issued by the same Synchronization Controller

In general to obtain a robust solution we should apply the following implementation rule:

> **Implementation Rule:** Any two Actions of the same resource are always issued by the same Synchronization Controllers.

This rule is applied in Chapter 4to design a robust implementation. If this rule is satisfied we are sure that no two Synchronization Controllers are ever interacting with the same resource at the same time. Notice that this rule poses a sufficient but not necessary condition to avoid the potential race conditions. In fact if one does not obey the rule, but can still guarantee that the Synchronization Controllers are never interacting simultaneously with the same resource, race conditions can not occur either.

3.4.3 Executable Model of Execution Framework

A detailed executable model of the Execution Framework expressed in POOSL [8]is created to get a deep understanding of the potential race condition and analyze it. The model is further used to show that the solution we design in Chapter 4 satisfies the Functional Specification, is robust and has the same performance as the original solution. The model refines a model that was

already available within ASML with components crucial to study the *Chuck Exchange Sequence*. This model is built using the Y-Chart approach [7]. This structure is reflected in the top level of the model containing an Application Cluster, a Mapper and a Platform cluster (see Figure 46). The entire Execution Framework is modelled in the Application Cluster. A snapshot of the Application Cluster is shown in Figure 47.

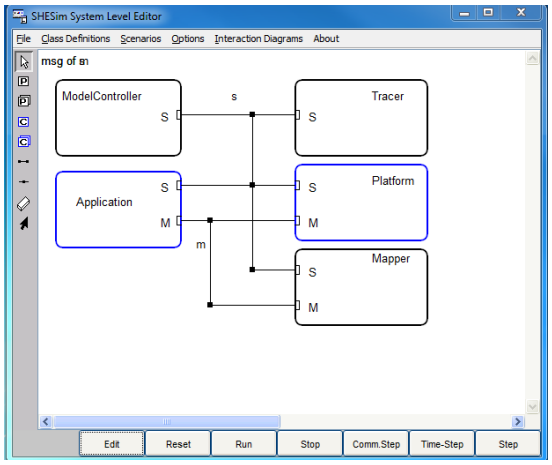


Figure 46: Top-level structure of the Execution Framework Model

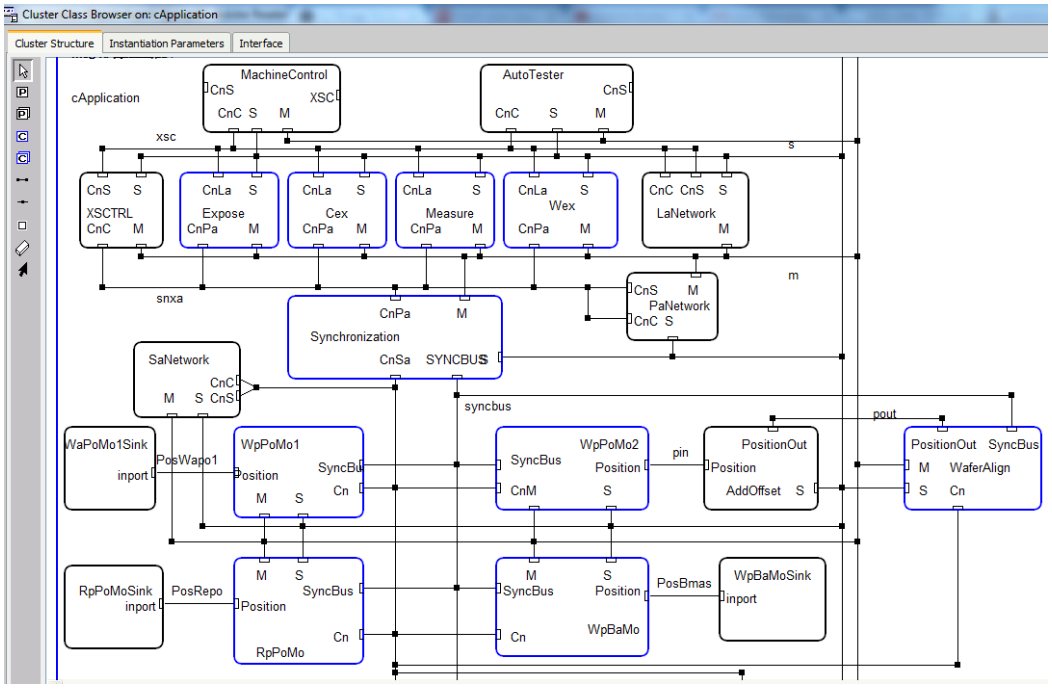


Figure 47: Application Cluster of the Execution Framework Model containing the layers of the Execution Framework

All the layers of the Execution Framework explained earlier are modelled in the Application Cluster of the executable model (see Figure 15). The analogy of the different layers in the model is as below in Figure 48.

The Chuck Exchange Controller corresponds to the MachineControl component, the Chuck Exchange Logical Action Component corresponds to the Cex component, the Synchronization Controllers corresponds to Synchronization component and the Resource layer corresponds to WpPoMo1, WpPoMo2, WpBaMo components. The Synchronization Cluster contains the two Synchronization Controllers similar to those in the actual Execution Framework which is shown in Figure 49.

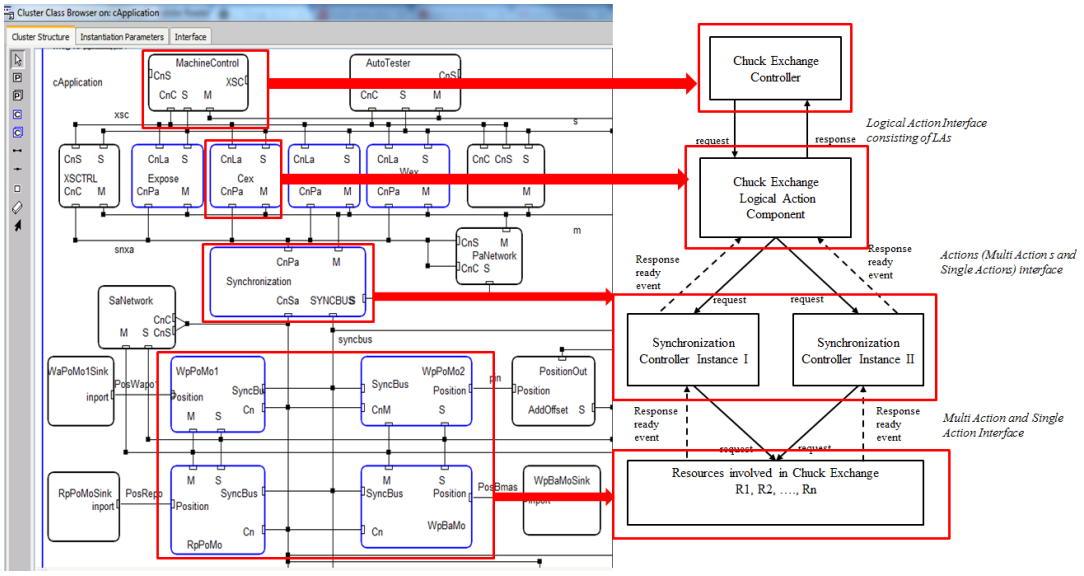


Figure 48: Analogy between executable Model of Execution Framework and the original Execution Framework

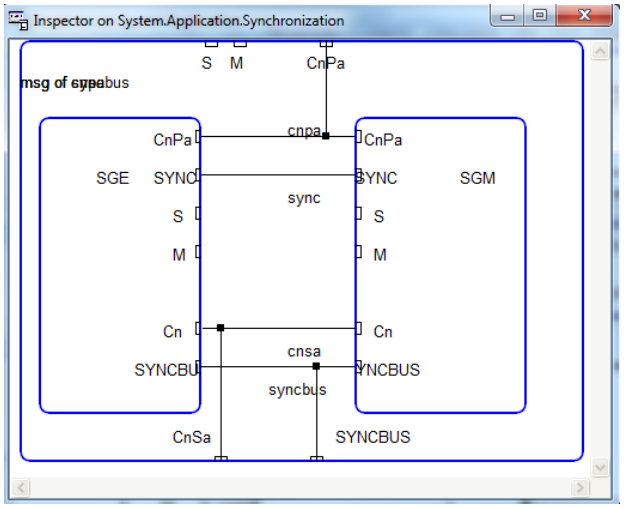


Figure 49: Two instantiations of Synchronization Controller in the executable model of the Execution Framework

The *Chuck Exchange Sequence* is implemented in the executable model of the Execution Framework. Due to timing variations in the model, a simulation run sometimes completes successfully and sometimes results in erroneous behavior. In the former case the Gantt chart generated by the executable model is compared with the one generated by the Functional Specification shown in Figure 14 of Chapter 2. The order of execution of the Actions at the resources in both the Gantt charts is same which proves that the model is consistent with the Functional Specification of the *Chuck Exchange Sequence.* The Gantt chart generated by the executable model is shown in Figure 50. The orange and pink bars differentiate the Start-Start and Finish-Start dependencies. The X axis shows timing in milliseconds and the Y axis shows the protocols that are implemented at each layer to execute a Logical Action. These protocols are nothing but the issuing of actions (Logical Action and their corresponding Actions) at each layer, their execution at the resources and retrieval of their results.

Figure 50: Gantt chart generated by the executable model of the Execution Framework

In case the potential race condition is triggered in the executable model the Gantt chart generated is not complete. In Chapter 4 the executable model is used to validate that the solution is robust indeed.

Summary

The focus of this chapter was the analysis of the potential race condition. To this end we showed in detail the required concepts used in the Execution Framework to implement the Functional Specification of the Chuck Exchange Sequence. In addition we made explicit the alternative choices to implement the functional dependencies (in the Functional Specification) in terms of these implementation concepts. To design robust (race condition free) sequences we formulated an Implementation Rule. This rule is used in the next chapter to design our robust solution. To validate this robust solution, we created a detailed executable model and verified its correctness against the Functional Specification.

Chapter 4

In this chapter a new Implementation Specification is designed that satisfies the Functional Specification (derived in Chapter 2) and is robust (based on the Implementation Rule that we formulated in Chapter 3). The design of the new Implementation Specification is carried out in two steps. In Section 4.1 the Implementation Rule discussed in Section 3.4.2 is enforced. The resulting Implementation Specification is robust but results in a performance penalty. To address this problem an adaption is made in Section 4.2 to obtain a robust solution without suffering from this performance penalty. The correctness of the new Implementation Specification is verified by comparing its Gantt charts to the Gantt chart of the original Functional Specification that was shown in Figure 41. In addition, its performance is compared to the performance of the original Implementation Specification and is shown to be the same. In 4.3, the potential occurrences of race conditions are analyzed. A summary is given in Section 4.4

4.1 Step 1

In order to enforce the Implemenation Rule seen in Section 3.4.2 on the original Implementation Specification, Single Action 9 is shifted to LA5 so that the same Synchronization Controller issues both the Actions (Single Action 5 and Single Action 9) to Resource 3. Thus, LA5 contains all its previous Actions along with Single Action 9 while LA7 contains Multi Action 7 and Multi Action8. Finish-Start and Start-Start dependencies in the original Chuck Exchange DAG (Figure 16) remain the same. The new Implementation Specification is shown in Figure 51.

The implementation of the functional dependencies in the new Implementation Specification is same as that explained in Section 3.4.1; all the functional dependencies in the original Functional Specification (shown in Figure 38) are respected. However, the new Implementation Specification shows an additional functional dependency between Single Action 9 and Multi Action 7 (see Figure 52) which is introduced because of the Finish-Start dependency between the new LA5 and LA7. This additional functional dependency results in a performance penalty. This is seen in the Gantt chart of the new Implementation Specification shown in Figure 53. Comparing this new Gantt chart with the Gantt chart of the original Implementation Specification (see Figure 42) shows that the make span on Resource 1 is larger in the new Implementation Specification. Thus, although, shifting the Single Action 9 to LA5 gives a functionally correct and robust Implementation Specification, it results in a performance penalty.

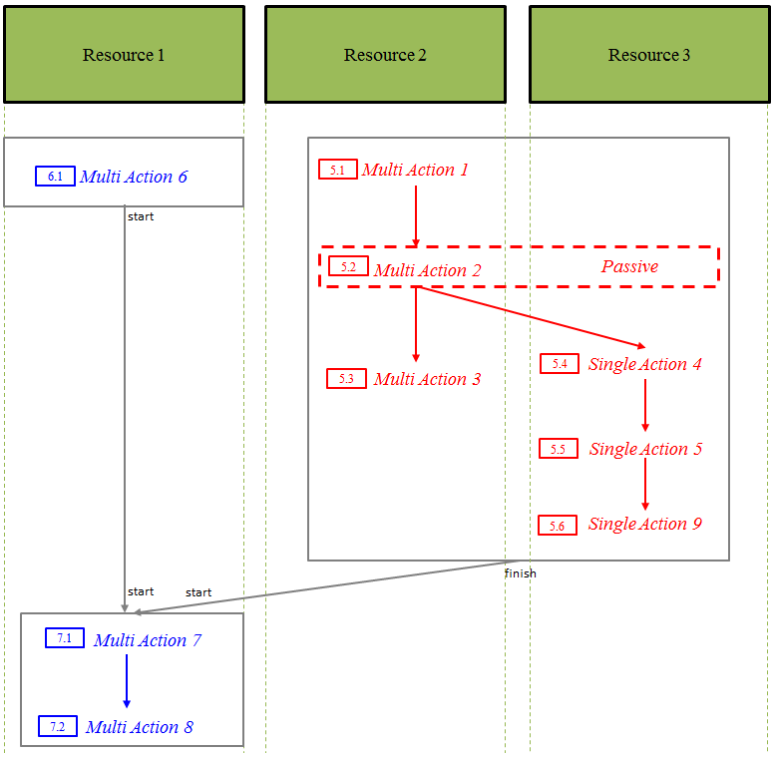


Figure 51: Step 1 in new Implementation Specification

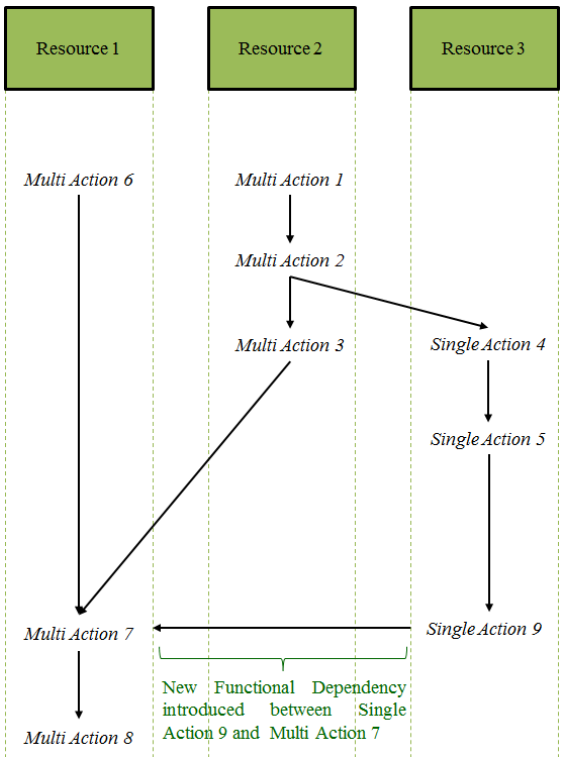


Figure 52: Step 1- Functional Dependencies introduced by the new Implementation Specification

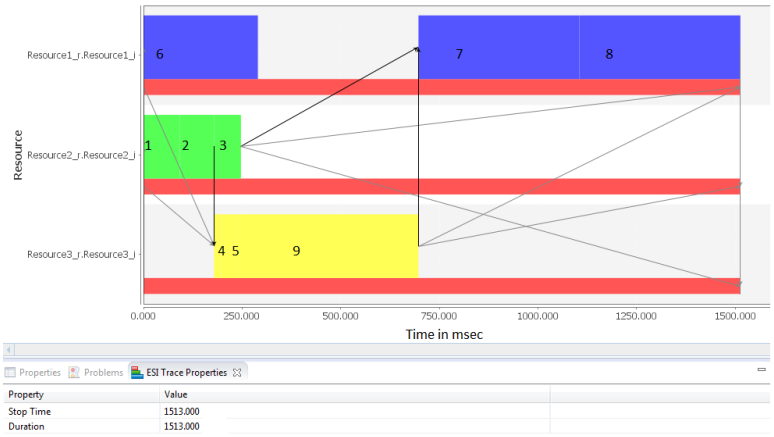


Figure 53: Step 1 - Gantt chart of the new Implementation Specification

4.2 Step 2

In this step, the Implementation Specification that was designed in Section 4.1 is further analyzed to improve its performance.

It is possible to divide LA5 into two separate Logical Actions, say LA5.a and LA5.b and address them to the same Synchronization Controller. The division of LA5 is such that the Logical Action LA5.a contains Multi Action 1, Multi Action 2 and Multi Action 3 while Logical Action LA5.b contains Single Action 4, Single Action 5 and Single Action 9. This results in a new DAG which is shown in Figure 54. The issuing order of Logical Actions at the Chuck Exchange Controller is now as follows:

$$LA1 \rightarrow LA2 \rightarrow LA3 \rightarrow LA4 \rightarrow LA5.a \rightarrow LA5.b \rightarrow LA6 \rightarrow LA7 \rightarrow LA8$$

A Start-Start dependency is created between the new Logical Actions, LA5.a and LA5.b and a Finish-Start dependency is enforced between LA5.a and LA7. The execution of LA7 and LA5.b is independent. The new Implementation Specification is shown in Figure 55.

The implementation of functional dependencies in this new Implementation Specification is as follows.

- The functional dependencies between Single Action 4 and Single Action 5 and between Single Action 5 and Single Action 9 are implemented by the issuing order of the Synchronization Controller to which all the Actions belong. Here, the implementation of rule 1 described in Section 3.3.1a is used. Since all the Actions belong to the same Logical Action, which is LA5.b, they are issued by the same Synchronization Controller.
- The functional dependency between Multi Action 2 and Single Action 4 is implemented by a Passive Claim on Resource 3 and by the issuing order of LA5.a and LA5.b. Here, the implementation of rule 4 described in Section 3.3.2a is used. See Figure 27 for implementation of rule 4.

- The implementation of the remaining functional dependencies is same as those explained previously in Section 3.4.1.

The new Implementation Specification obeys all the functional dependencies that are specified in the original Functional Specification shown in Figure 38. Thus, the new Implementation is functionally correct. It is robust as it respects the Implementation Rule discussed in Section 3.4.2. In the next section we analyze the performance of the new Implementation Specification.

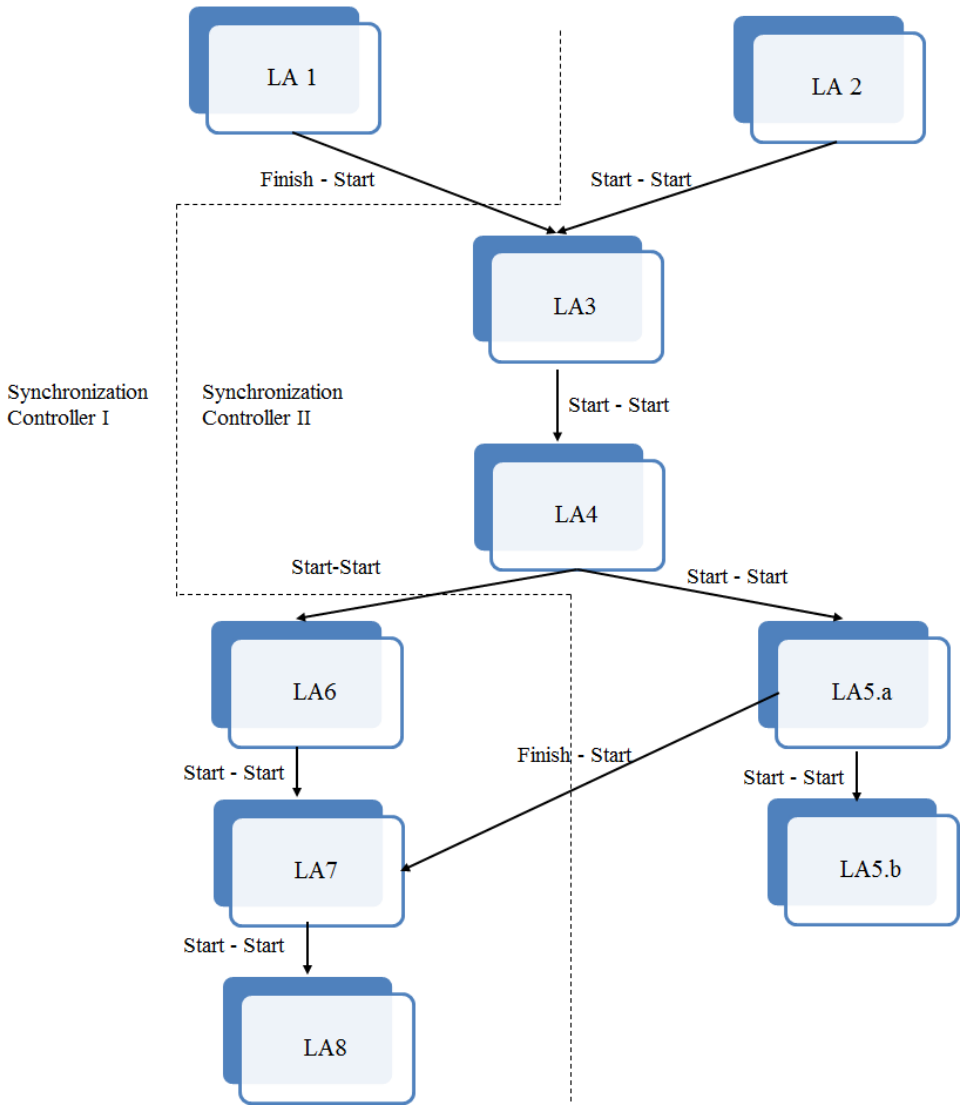


Figure 54: New DAG for the execution of *Chuck Exchange Sequence* at Chuck Exchange Controller

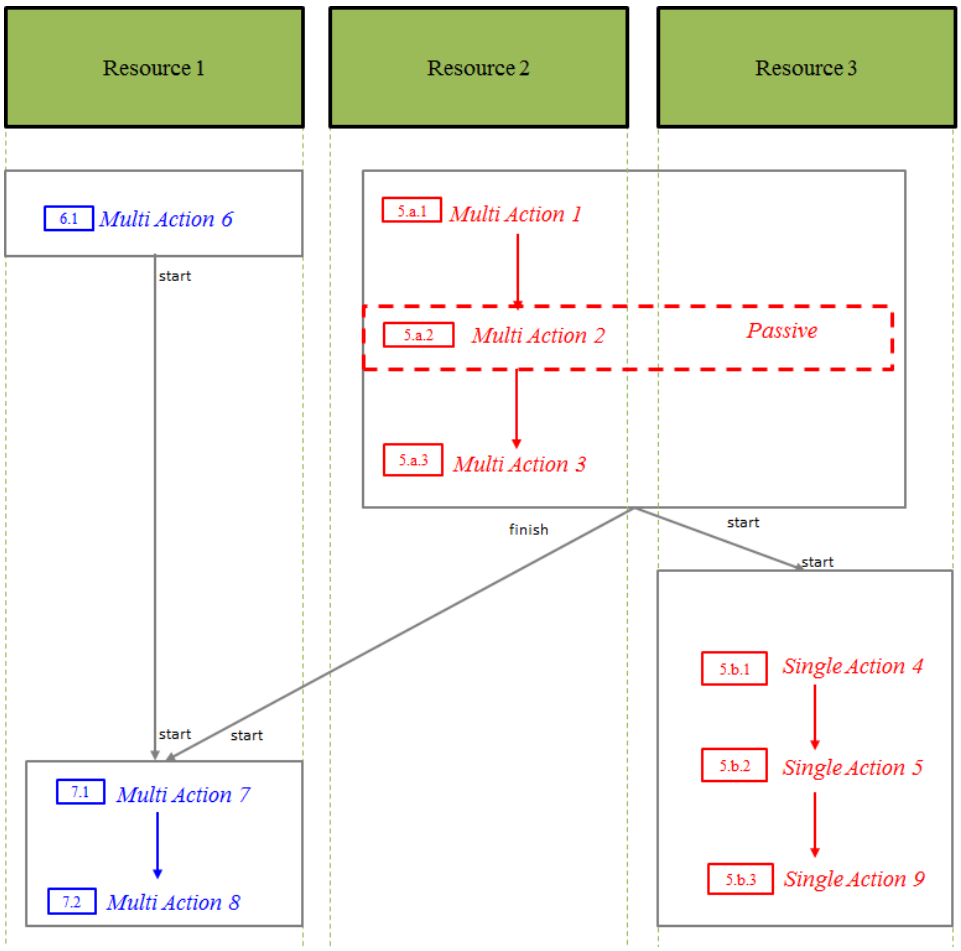


Figure 55: Step 2- New Implementation Specification

4.2.1 Performance Verification of the improved Implementation Specification

The Gantt charts of the new Implementation Specification (shown in Figure 56) and the original Implementation Specification (shown in Figure 42) are equivalent. This specification however only concerns part of the *Chuck Exchange Sequence*. In order to verify the performance, we need to compare the Gantt charts of the complete sequence. These Gantt charts are shown in Figures 57 and 58. The red blocks in the schedules show the part of the *Chuck Exchange Sequence* where the race condition might occur. Both the Gantt charts show the same make span per resource and therefore, the overall performance is the same. The new Implementation Specification is implemented in the executable model and it is tested for robustness. The execution of the *Chuck Exchange Sequence* completes in every simulation run, as expected.

Hence, we conclude that the new Implementation Specification is functionally correct and robust and results in same performance as the original Implementation Specification.

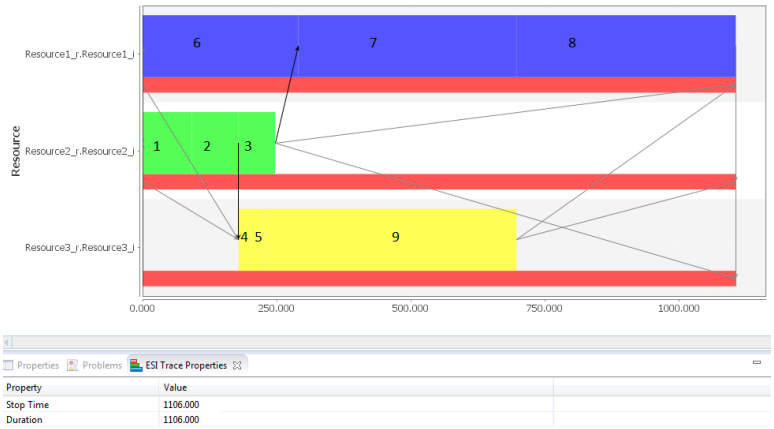


Figure 56: Step 2 - Gantt chart of the new Implementation Specification

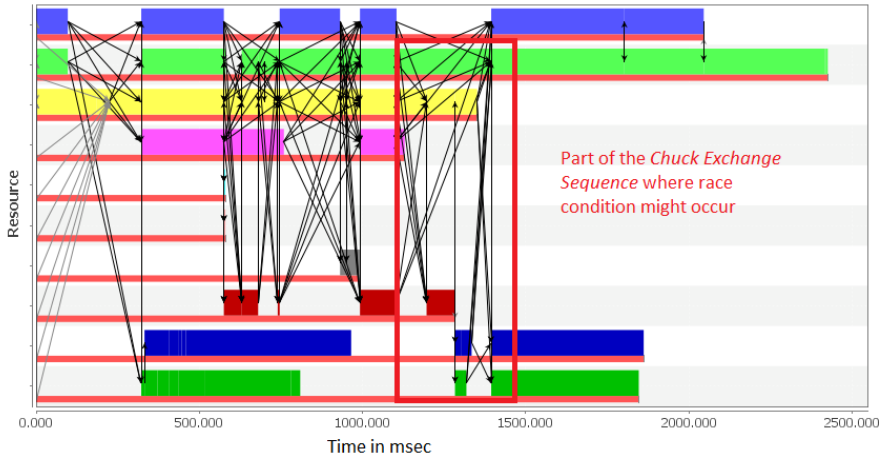


Figure 57: Gantt chart of Original Implementation Specification of the complete *Chuck Exchange Sequence*

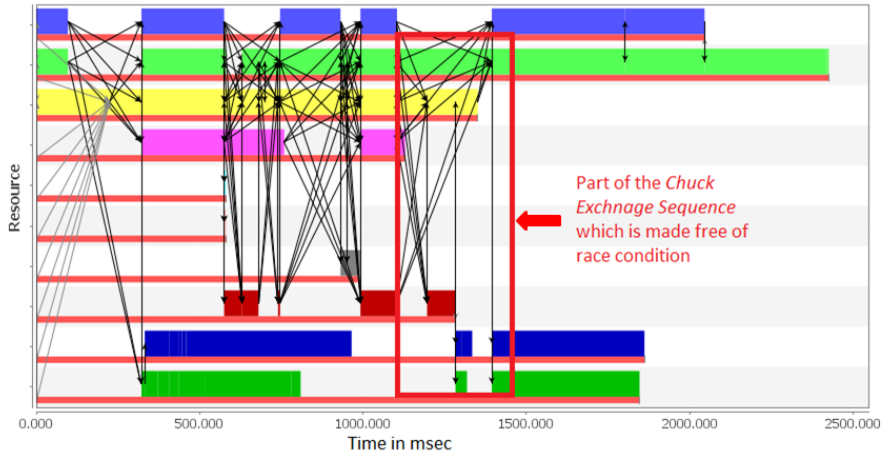


Figure 58: Gantt chart of New Implementation Specification of the complete Chuck Exchange Sequence

4.3 Potential other occurrences of Race Conditions

The new Implementation Specification eliminates the possibility of race condition in the part of the *Chuck Exchange Sequence* that is shown in Figure 39. In order to avoid all the potential occurrences of race conditions in the *Chuck Exchange Sequence*, the Implementation Rule mentioned in the Section 3.4.2 should be strictly obeyed. The analysis of the entire *Chuck Exchange Sequence* shows two other locations in the sequence at which this rule is not satisfied. However, race conditions are not likely to occur at these locations the Actions are separated in time. Also due to time limitations in our project we did not find a way to strictly apply our Implementation Rule while preserving (or potentially even improving) the performance. Notice that this does not imply that such a solution does not exist. In Chapter 5 we will show that the design space is huge and impossible to explore completely by hand.

4.4 Summary

In this chapter, the steps taken to derive an improved Implementation Specification for the *Chuck Exchange Sequence* were discussed. This Implementation Specification was shown to be functionally correct as well as robust. It was also shown that the execution of the improved Implementation Specification gives the same performance as that of the original Implementation Specification for the complete *Chuck Exchange Sequence*.

Chapter 5

In this chapter we present the conclusion of the work done along with the possibility for future extension of this work.

5.1 Conclusion

This project provides a robust solution to avoid a potential race condition in the *Chuck Exchange Sequence* of an ASML TWINSCAN Lithography machine, having the potential to improve system availability. The solution is in terms of a new Implementation Specification that is shown to be robust, functionally correct and on par with respect to performance to the original Implementation Specification.

The solution is derived in a model-based fashion in several main steps:

1. The implementation of the *Chuck Exchange Sequence* is revered-engineered and a Functional Specification (refraining from any implementation details) is derived. The concepts required to describe the sequence at the functional level are defined.
2. The key concepts to implement the sequence in terms of the available Execution Framework are defined. The current implementation of the *Chuck Exchange Sequence* is expressed in terms of these key concepts, resulting in an Implementation Specification.
3. A detailed executable model of the Execution Framework is made to get insight in the implementation and to analyze in detail the current and improved implementations of the sequence.
4. The current executable model and the Implementation Specification are used discover the location at which the potential race condition can occur and explain the reason why it can occur.
5. An Implementation Rule is derived that guarantees the absence of race conditions.
6. An improved Implementation Specification is designed, based on the Implementation Rule which satisfies the Functional Specification, is robust and preserves performance.
7. The improved Implementation Specification is implemented in the executable model to carry out a detailed analysis of the new solution through simulation, thereby increasing the evidence of the correctness of the solution.

5. Future research

The steps taken in this work to obtain a robust implementation of the sequence are made manually. The number of possible implementations for each Functional Specification is huge however:

- Actions can be grouped in many ways in Logical Actions. In fact the number of possible partitions of a set of Actions is given by the so-called Bell number which is defined by

the following recurrence equation$B(n) = \sum_{k=0}^{n} \binom{n}{k} B(k)$, where n denotes the number of actions.

- For each pair of Logical Actions, a choice has to be made whether they have a Start-Start or Finish-Start relation.
- For every logical action a choice has to be made on the Synchronization Controller that issues its constituent Actions.
- The issuing order for the Logical Actions has to be decided upon.
- The issuing order of the Actions internal to each Logical Action has to be decided upon.
- It has to be decided at which positions to introduce passive claims.

Obviously, in our manual exploration we only have been able to explore a few implementation alternatives and luckily we have been able to discover an appropriate one that eliminates one potential critical race condition occurrence. On the other hand, we have not succeeded to find a solution eliminating all potential race conditions, while preserving performance.

A very interesting topic for future research is therefore to completely automate the design-space exploration, see Figure 59. The idea is to start with a fully Functional Specification and then derive a correct and robust Implementation Specification with (near) optimal performance in a fully automated way. Most important advantage would be that it relieves the software designer to come up with such solution, which we have experienced to be complex and error-prone. To reach this automation goal, the concepts defined in this report have to be fully formalized.

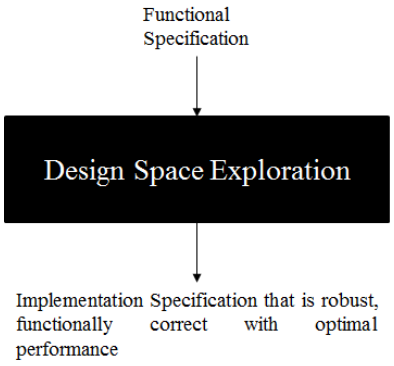


Figure 59: Automated Design Space Exploration

References

[1] Avanti Vyas; Voeten J.P.M; Wouter Tabingh Suermondt, *'Design Space Exploration for Chuck Exchange',* Master Thesis Preparation Phase Report, ASML, Veldhoven, 2015.

[2] Avanti Vyas, '*Functional Specification for Chuck Exchange*', ASML, Veldhoven, 2015.

[3] ASML, *'NT2 1950 WS chuck exchange',* ASML, Veldhoven, 2011.

[4] Putten, P.H.A. van der; Voeten, J.P.M*, 'Specification of Reactive Hardware/Software Systems – The Method Software/Hardware Engineering',* University of Technology, Eindhoven, The Netherlands, 1997, ISBN 90-386-0280-4ICS, EB.

[5] Fangyi Shi, *'Wafer Logistics Specification Tool User Manual',* ASML, Veldhoven, 2015.

[6] Wilbert Alberts, *'SDS Software Synchronization Control*', ASML, Veldhoven, 2000.

[7] J. Voeten; T. Hendriks; B. Theelen; J. Schuddemat; W. Tabingh Suermondt; J. Gemei; K. Kotterink; C. van Huet, *'Predicting Timing Performance of Advanced Mechatronics Control Systems',* 35th IEEE Annual Computer Software and Applications Conference Workshops (COMPSACW), 2011.

[8] M.C.W. Geilen; J.P.M. Voeten; P.H.A. van der Putten; L.J. van Bokhoven; M.P.J. Stevens, *'Object-oriented modelling and specification using SHE'* Section of Information and Communication Systems, Faculty of Electrical Engineering, University of Technology, Eindhoven, The Netherlands, 2001.