An analysis of ASD timers and their expressivity with mCRL2

Hong, T.

Award date:
2019

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

An Analysis of ASD Timers and Their Expressivity with mCRL2

Master Thesis

Tao Hong

Supervisors:
Jan Friso Groote
Thomas Neele

Public version

Eindhoven, February 2019

Abstract

For model-driven engineering projects in ASML, the tool ASD is used, where ASD is a control certification framework. ASML has developed a model conversion system that can translate ASD models into mCRL2 models. Because more and more requirements have to be formally verified in ASML, it is necessary to expand the existing model transformation system to support real-time models.

Real-time modelling in ASD is achieved by using the timer service. However, after the model transformation, timing information contained in the ASD timer service is omitted from the mCRL2 model. This project aims at analysing the ASD timer service and proposes a way to express ASD timers in mCRL2 for ASD timer services. At the same time, the proposed timer expressions in mCRL2 should be capable of expressing logic relationships contained by real-time requirements from ASML properly.

A simple ASD model is first implemented and translated into an mCRL2 model. By fixing the transformed mCRL2 model manually, problems exist when dealing with the real-time models are located and further modifications to improve the model transformation system are proposed.

Subsequently, some representative ASML real-time requirements are gathered, and mCRL2 models are formulated based on those requirements to demonstrate the suitability of mCRL2 timer processes. In the implemented mCRL2 models, ASD timer services are expressed by timer processes which have the same mechanism as the ASD timer service.

The properties of the obtained timed models have been analysed. However, during the analysation, limited by the mCRL2 real-time model verification tools, compromises had to be made. That could be a challenge when applying modifications about real-time models to the current model transformation system.

Contents

Chapter 1

Introduction

In wafer production machines, real-time systems play an essential role. When designing a lithography machine, time constraints are strict: tasks need to be assigned precisely so that various functions can work closely together. As a benefit of proper time scheduling, the throughput of lithography machines is increased, thereby decreasing their running costs.

For example, in a lithography machine, wafers are first scanned to collect data. Then, based on the received data, wafers can be exposed. It is required that after the information is collected, the waiting time of a wafer should not exceed a certain amount of time; otherwise, deformation may be caused by small temperature changes. The accuracy of the subsequent exposure process will be affected by those unwanted deformations. Due to this requirement, the behaviour of machine components needs to be controlled to take place at appropriate times. At the software level, the behaviour of each machine component is the result of running thousands of tasks.

To solve such complex situations, a model can be implemented first, the entire software development process is then based on this model. Such an approach is called model-driven engineering. It requires to formulate behaviours of software systems by using models before implementations. According to the research of [1], with the verified models, there are fewer bugs in complex software. Based on the concept of model-driven engineering, modelling in ASML relies on the ASD: Suite [2]. ASD:Suite is a component-based modelling formalism for software design from Verum, which uses *interface models* for verification and *design models* for implementation. ASML has developed a transformation system, which is capable of transforming ASD models into labelled transition systems on which more properties can be verified.

This model transformation system contains three components: ASD, ALIAS and MIDS, where ALIAS is used as a control specification language designed by ASML. It can be used to translate ASD models into technology independent control models. With MIDS, ALIAS models can be further converted into mCRL2 models. mCRL2 is a formal specification language which can be used for modelling and validation. When time properties are involved in the ASD model, built-in timer services are used to restrict the behaviours related to time. However, in the current transformation system, timing info is omitted while transforming models, and that causes changes in the behaviour of real-time models.

The goal of this project is to explore the possibility of transforming real-time ASD models into mCRL2 while preserving timing information. To achieve this, the ASD timer service is analysed first to find what can be improved in the current transformation system when dealing with real-time models. Then, requirements related to time from ASML projects are inspected and classified. Based on ASML real-time requirements, mCRL2 models are implemented with timer processes which have the same working principle as the ASD timer service. Those mCRL2 models are then verified to prove the applicability and scalability of the ASD timer service.

Backgrounds of ASD, ALIAS, MIDS and mCRL2 are first explained in Chapter 2. Then the timer service is explored, and the potential of making use of this service is discussed combined with ASML real-time requirements in Chapter 3. Chapter 4 explains some examples of using timer services in mCRL2 models. Besides mCRL2, there are also other tools that are capable of dealing

with real-time models. Those tools are mentioned in Chapter 5. We conclude in Chapter 6.

Chapter 2

Preliminaries

In this chapter the working principles of ASD, ALIAS, MIDS and mCRL2 are explained.

2.1 ASD

Analytical Software Design (ASD) [2, 3] is a control certification framework from Verum. It is a model-driven software engineering tool to create, explore, and formally verify component-based designs for embedded and technical software systems. Two kinds of models can be built with ASD. A *design model* describes the internal behaviour, and an *interface model* defines externally visible behaviour (the expected behaviour).

In ASD models, a designer can control the occurrences of actions by using Sequence-Based Specification (SBS) rules. Once the design model and the interface model are implemented, the tool-set allows the designer to verify properties like the absence of deadlocks or illegal actions.

ASD is a component-based technology, and each ASD model can contain multiple components. Each of those components includes states and transitions with events and actions to describe the behaviour of the component. All ASD components must have both an interface model and a design model. In the following subsections, more details about SBS rules and components are explained.

2.1.1 Interface model and design model

The interface model is a model of the externally visible behaviour of an ASD component without implementation details. This means that an interface model describes how a component should behave under every circumstance other than how the component achieves such behaviours. The external behaviour is specified independently of any specific implementation. It is an abstraction of designs.

The design model implements the internal behaviour of the interface model. Different from interface models, design models contain more details, and they can use lower level components through their interface models.

ASD components are software components specified and designed using ASD. An ASD component contains an interface model that communicates with higher level components called servers. A design model describes the model details. Interfaces that are implemented by lower level components which are called clients. Each ASD component implements a *service* that can be used by its clients. Clients access the service by sending *call events*. Then the ASD component can respond to this call event through an application interface with a *reply event* and a *notification event*.

Information passed to a component is indicated by *triggers*. A trigger can be a call event from a client, a reply event from a server or a notification event from a server. A component exposes its information to its clients and servers with *actions*. An action can be a call event to a server, a reply event to a client or a notification event to a client.

2.1.2 Sequence Based Specifications

ASD models are implemented as State Machines. A state machine contains transitions and states. In ASD design models, a transition is defined using Sequence-Based Specification (SBS) rules. An SBS rule consists of a Current State, a Trigger, a Guard, Actions, Updates and a Target State. This rule describes the event, actions and how the target state is reached from the current state.

An example of SBS rules is presented in Figure 2.1. The target state indicates the next state of the model if the transition is available (not illegal nor blocked). The flow of the component can be controlled by using guards and updates on state variables.

Interface	Event	Guard	Actions	State Variable Updates	Target State
NotActivated (initial state)					
	StateInvariant		-		-
IAlarmSystem	switchOn+		IAlarmSystem.OK		Activated_Idle
IAlarmSystem	switchOff		Illegal		-
Internal	[Triggered]		Disabled		-
Internal	SwitchedOff		Disabled		-
Activated_Idle					
	StateInvariant		-		-
IAlarmSystem	switchOn+		Illegal		-
IAlarmSystem	switchOff		IAlarmSystem.VoidReply		Deactivating
Internal	[Triggered]		IAlarmSystem_CB.triggered		Activated_AlarmMode
Internal	SwitchedOff		Disabled		-
Deactivating					
	StateInvariant		-		-
IAlarmSystem	switchOn+		Illegal		-
IAlarmSystem	switchOff		Illegal		-
Internal	[Triggered]		Disabled		-
Internal	SwitchedOff		IAlarmSystem_CB.switchOff_cb_OK		NotActivated
Activated_AlarmMode					
	StateInvariant		-		-
IAlarmSystem	switchOn+		Illegal		-
IAlarmSystem	switchOff		IAlarmSystem.VoidReply		Deactivating
Internal	[Triggered]		Disabled		-
Internal	SwitchedOff		Disabled		-

Figure 2.1: SBS rules of the alarm system design model

Actions can be used to invoke operations on lower level components. In this situation, a method call requests lower level components to perform specific actions and notifications to indicate that the lower level component has finished its task. Upon receiving a method call, it is required to generate a reply to inform the caller that it has finished processing the method call. The replied value can either be valued or *void*. Furthermore, actions can be used to send notifications from lower level components to higher level components. The purpose of notifications is to signal the higher level components that the lower level component has finished the assigned tasks.

A complete series of operations consists of method calls, notifications and a reply if the event is a method call. After sending a method call, the caller components will temporarily be blocked until it receives a reply from the callee. Different from method calls, notifications are sent to a queue, and this queue has a higher priority than a new method call from the client using the component. When the queue is empty, the client can perform a new method call.

Events in ASD could be method calls from higher level components; notifications received from lower level components using interface, or modelling events which describe internal behaviours of the component implementing the interface. In the ASD interface model, those events are defined in every state, and the user has to specify whether an event is legal, illegal or being blocked in each state.

If the event is legal, depending on the internal state of the interface, the user can further define whether a specific event will take place or no operation will take place at all. Furthermore, if the event is legal, a reply will be sent back to the client that uses the interface.

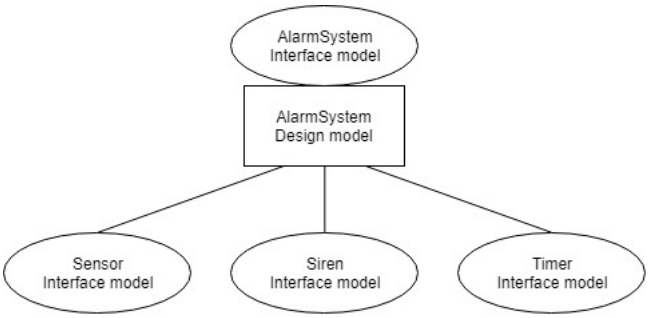


Figure 2.2: Interface and design models of the alarm system

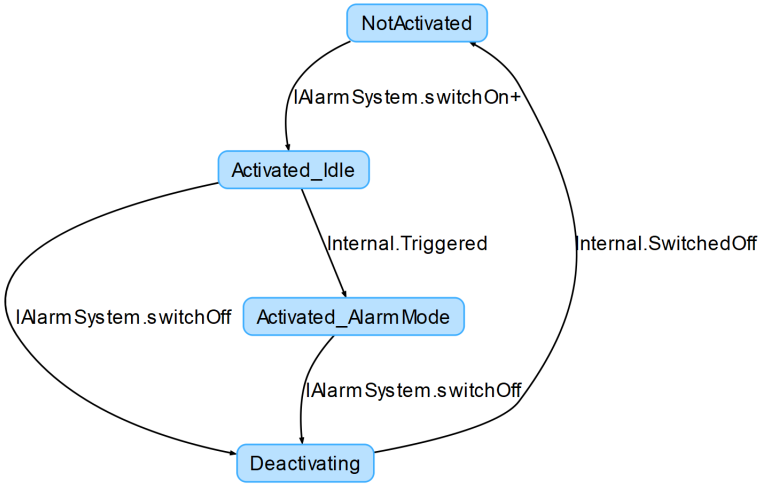


Figure 2.3: The state space of the alarm system interface model

The alarm system presented in Figure 2.1 is used as an example to explain the mechanism of ASD. In Figure 2.2, a schematic view of an alarm system is presented. In this system, the interface model "AlarmSystem Interface model"contains four states: *NotActivated*, *Activated_Idle*, *Deactivating* and *Activated_AlarmMode*. The state space of the interface model is presented in Fig 2.3. In the initial state, the alarm system is *NotActivated*. After the event *SwitchOn* it transfers to *Activated_Idle*. In the state *Activated_Idle*, the system can either go to *Deactivating* with the *SwitchOff* event when the user turns the system off before the alarm is triggered, or the alarm is triggered by the action *AlarmTripped*. In the latter sense, the state will change to the state *Activated_AlarmMode*. In the state *Activated_AlarmMode*, the alarm system can be turned off with the action *SwitchOff* and switches to state *Deactivating*. From the state *Deactivating*, the event *SwitchedOff* indicates the system has been turned off and went back to the initial state.

Figure 2.4 indicates the state space of the design model. In the design model of the alarm system, three more components are used: Sensor, Siren and Timer. Those components are called through their interface models. After the alarm system is turned on the sensor will be activated. The system can either be switched back off, and the sensor will be turned off, or movement is detected by the sensor, and a notification is sent to the alarm system.

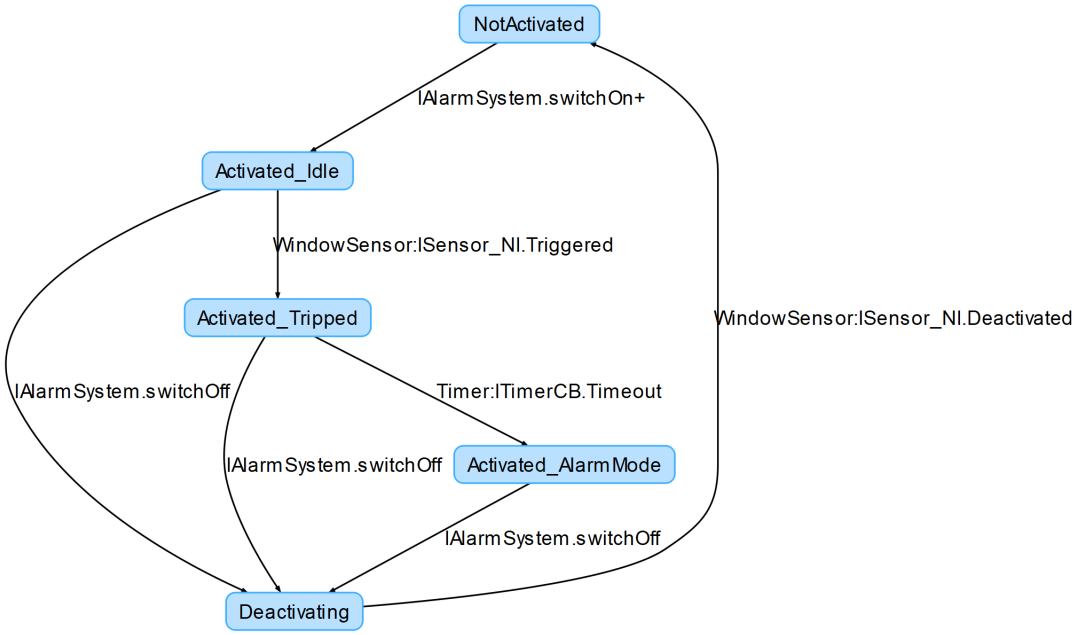


Figure 2.4: The state space of the alarm system design model

Compared with the interface model, the design model contains one more state called *Activated_Tripped*. The state *Activated_Tripped* can be reached if the sensor is triggered. In this state, the timer will start counting, and the siren remains turned off. After the time is over, the siren will be turned on. The siren and the timer can be turned off with the action *SwitchedOff*. In this example, the sensor, siren and timer are controlled by the design model of the alarm system, where it models the internal behaviour of the alarm system, coordinating those three components to make the design model consistent with the interface model.

2.1.3 Verification

Once the design model and interface model are formulated, they can be verified by the ASD: Suite. The verification of design models also includes its referenced interface models, which ensures the behaviour specification is complete and the component behaves correctly in its environment. With the mechanism that components communicate with each other through interface models, the correctness of the entire system can be ensured by verifying each component individually.

With the verification, the ASD: Suite ensures that interface models and design models are complete and well formed, range errors, live-locks and deadlocks are absent. Furthermore, the ASD: Suite assures interface compliance of models. For example, the behaviours allowed in the interface model should also be permitted by the design model, and the design model should not refuse to do actions that the design model requires it to do. Besides, the verification also ensures the design models comply with the service specifications of the components it uses. It is noteworthy that timing properties in a real-time system will not be verified by the ASD: Suite.

2.2 ALIAS

ALIAS is a control specification language as a part of the ASML Software Modeling Environment (ASOME). The purpose of ALIAS is to create technology independent control models while using ASD as the control verification framework. Furthermore, ALIAS connects control models to the

data and algorithm models. Besides, ALIAS is used to generate the glue code to embed the control code into ASML code automatically.

In this project, the primary function of ALIAS is to transform ASD models into ALIAS. The translated ALIAS code keeps all the states and transitions of the ASD model. With the tool MIDS, ALIAS code can be converted into an mCRL2 specification. The details of ALIAS will not be included in this report since it has similar concepts as ASD.

2.3 mCRL2

mCRL2 stands for micro Common Representation Language 2, and it is based on the Algebra of Communicating Processes [4] extended with data and time. It describes sequences of actions with process algebra. In this section, an introduction of mCRL2 is given.

2.3.1 Grammar

States and transitions in ASD are translated into processes in mCRL2. The operators used in the translation from ASD to mCRL2 is given below:

P =

a	Action
Q	Process
P+P	The choice operator
P.P	The sequential operator
P|P	The communicate operator
P‖P	The parallel operator
(g)->P	If statement
sum x:D. P	The summation operator
(P)	Brackets
$P^{\mathsf{c}}t$	Time

where g is a Boolean expression and x is data variable, D is the data type defined internally in mCRL2. Data type natural numbers $\mathbb{N}$ and data type real numbers $\mathbb{R}$ are data types commonly used in this project. The choice operator indicates that a non-deterministic choice can be made between the left-hand expression and the right-hand expression. The sequential operator makes the actions take place in sequence from left to right. The communication operator allows multiple actions to take place at the same time and the parallel operator allows multiple processes to execute at once. In the if statement, the execution of the process P relies on the boolean expression g. The process P can be processed only if g is true. The summation operator is a generalization of the choice operator: it generates variable d of data type D, the result of the summation is $p[d_0/d] + \cdot + p[d_n/d], n \geq 0$, for all elements $d_i \in D$. Here, $p[d_i/d]$ stands for p in which each free occurrence of d is replaced by d_i. Summation has the lowest precedence after the choice operator.

Time expressions in mCRL2 are realized by the c operator. The following expression means that action a takes place at time t:

$$a^{\mathsf{c}}t$$

where t is a real number. The timed process behavior starts at time 0. An action can only take place at a time larger than 0. Furthermore, in timed processes, actions must take place in the right order according to their time tag. For example, in the process $a^{\mathsf{c}}1.b^{\mathsf{c}}3.c^{\mathsf{c}}2$, the action c cannot take place, and causes a deadlock at time 3.

By making use of the sum operator and the if-statement, the following expression can be used to indicate after an action a, action b must take place within s time units:

$$\textstyle\sum_{t:\mathbb{R}} a^{\mathsf{c}}t. \sum_{u:\mathbb{R}}(u \leq t + s) \rightarrow b^{\mathsf{c}}u$$

Where s is a positive real number. For a full description of the mCRL2 grammar, see [5].

2.3.2 The mCRL2 Toolset

The mCRL2 toolset provides over 60 tools for analyzing mCRL2 models. In this project, tools are used to generate state spaces and verify properties of mCRL2 models. Those functions rely on two types of objects: *linear processes* [6] and *parameterised Boolean equation systems* [7].

To verify the properties of mCRL2 models, mCRL2 specifications need to be encoded into a linear process. This transformation is done by the mCRL2 tool *mcrl22lps*. A linear process is an mCRL2 process specification with a restricted grammar. It is basically a syntactic format of a single-step transition relationship caused by a process [8]. Via linear processes, mCRL2 models can be simulated, and their state space can be generated and stored.

To verify an mCRL2 model, *Parameterised Boolean Equation Systems* (PBESs) [7] and formulas specified in the modal μ-calculus [9] are used. PBESs are generated with a linear process and a modal formula, where modal formulas are properties described with fixpoint equations [10]. Solutions of PBESs indicate whether behaviours described in model formulas hold.

2.3.3 Translation From ASD Models to mCRL2 Models

As mentioned before, in ASD a transition is characterized as an SBS rule. However, in mCRL2 models, transitions can only be specified by actions, each transition connects two states. In order to describe ASD models with mCRL2 languages properly, multiple internal states called *pseudo states* are introduced to connect the *Current state* and the *Target state*. *Trigger* and *Actions* of SBS rule is presented as actions in mCRL2. Those introduced internal states are called *pseudo states* [11].

Current State	Trigger	Guard	Actions	Updates	Target State
S1	Call_B	x==0	VoidReply_B; Call_D;Call_E	x = 1	S2

Table 2.1: An example of ASD rule case configuration

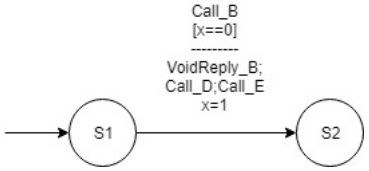


Figure 2.5: ASD rule case

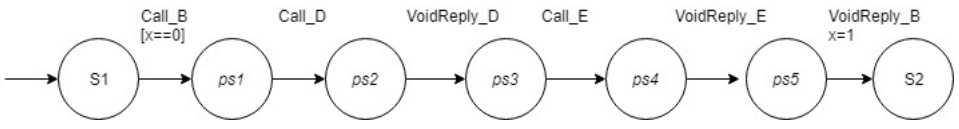


Figure 2.6: Translated ASD rule case in mCRL2

Table 2.1 indicates an example of an ASD rule case configuration. S1 is the initial state. If action Call_B takes place and variable x equals 0, then a series of actions VoidReply_B, Call_D and Call_E will take place. Target state S2 is reached, and variable x is updated to 1. Figure 2.5 is the state space of this model and Figure 2.6 presents the same behaviour translated by MIDS. As

can be seen, in the mCRL2 model, all the ASD trigger event and actions are split into individual transitions, and those individual behaviours are connected in series using pseudo states.

The guard of the ASD model is appended to the trigger event action in mCRL2 model and updates are associated with the last event action. Besides, for each of the call action (Call_D, Call_E), a reply action is added in the mCRL2 model. Thus, the caller component will wait for the callee to complete its execution which is an implicit behaviour in ASD.

Chapter 3

Timing in ASD

In the ASD Model Builder tool, a built-in service caller *Timer* is used to implement the model with timing requirements, and the Timer could be utilised for adding timing to ALIAS models. In this chapter, an ASD model using timer service is translated into an mCRL2 model. The behaviour of the translated ASD timer service in the mCRL2 model is then analysed.

3.1 The ASD Timer

The function of the timer service is to delay action for a user-defined period. The timer service is a built-in service, which means users cannot modify it directly in ASD. As with other ASD services, i.e. foreign components, the ASD timer implement triggers to indicate the start and the end of a timer. After a timer of x seconds is created with trigger *CreateTimer(x)*, the environment will start counting down x seconds. After x seconds have passed, the *TimeOut* event will take place. As long as the *TimeOut* did not happen, the timer can be canceled using the *CancelTrigger* and the *TimeOut* trigger will never occur.

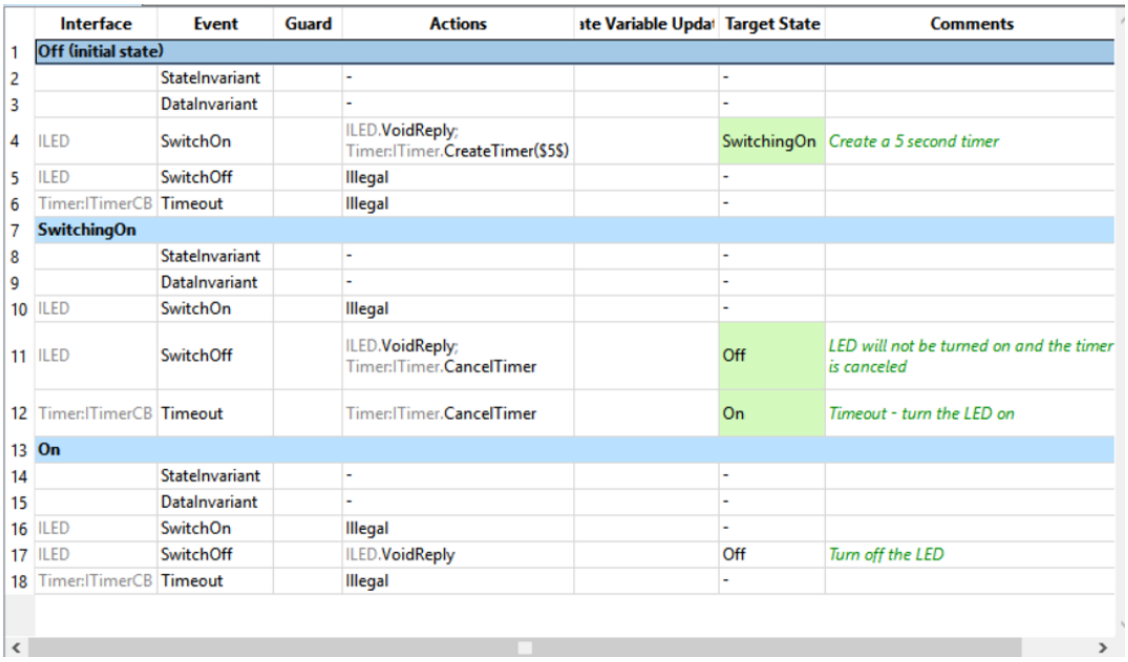
To research the behaviour of an ASD Timer in the translated mCRL2 model, a simple LED model is considered. The LED contains a *SwitchOn* button and *SwitchOff* button. In the initial state, the LED is off, and after the *SwitchOn* button is pressed, the Timer will be triggered. When the time is out, the LED light will turn on. Whenever the *SwitchOff* button is pressed, the timer will be cancelled, and the LED will switch off. The specification of the ASD interface and the ASD implementation are presented in Figure A.1 and Figure A.2.

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments
1	**Off (initial state)**						
2		StateInvariant		-		-	
3	ILED	SwitchOn		ILED.VoidReply		On	*LED is turned on*
4	ILED	SwitchOff		Illegal		-	
5	**On**						
6		StateInvariant		-		-	
7	ILED	SwitchOn		Illegal		-	
8	ILED	SwitchOff		ILED.VoidReply		Off	*LED is turned off*

Figure 3.1: The specification of an LED system

The ASD model of the LED is then transformed into an mCRL2 model using ALIAS and MIDS. As expected, since MIDS does not support time, the resulting mCRL2 model is not complete. Therefore, the transformed model is manually edited to support the timer, List 3.1 indicates the modified timer process in mCRL2, the whole mCRL2 model can be found in APPENDIX A.

In the mCRL2 model transformed by MIDS, the states in ASD implementation model are defined as processes, and so are the states of the ASD Timer. The timer in the mCRL2 model con-



	Interface	Event	Guard	Actions	ıte Variable Updat	Target State	Comments
1	**Off (initial state)**						
2		StateInvariant		-		-	
3		DataInvariant		-		-	
4	ILED	SwitchOn		ILED.VoidReply; Timer:ITimer.CreateTimer(5)		SwitchingOn	*Create a 5 second timer*
5	ILED	SwitchOff		Illegal		-	
6	Timer:ITimerCB	Timeout		Illegal		-	
7	**SwitchingOn**						
8		StateInvariant		-		-	
9		DataInvariant		-		-	
10	ILED	SwitchOn		Illegal		-	
11	ILED	SwitchOff		ILED.VoidReply; Timer:ITimer.CancelTimer		Off	*LED will not be turned on and the timer is canceled*
12	Timer:ITimerCB	Timeout		Timer:ITimer.CancelTimer		On	*Timeout - turn the LED on*
13	**On**						
14		StateInvariant		-		-	
15		DataInvariant		-		-	
16	ILED	SwitchOn		Illegal		-	
17	ILED	SwitchOff		ILED.VoidReply		Off	*Turn off the LED*
18	Timer:ITimerCB	Timeout		Illegal		-	

Figure 3.2: The implementation of an LED system

tains two processes: “ItimerProtocol’ Active” and “ItimerProtocol’ Inactive” to indicate whether the timer is active. In the initial state, the timer is inactive; The action “invoked”asomeITimer’ ITimer’Create” activates the timer. The action communicates with other processes with the action “invoke”asomeITimer’ITimer’Create” to check if the timer is created. In the LED example, it communicates with the “led’Off” process since it is the initial state and at this state, the timer is not enabled.

After the timer is enabled, the process “ItimerProtocol’Inactive” will send a reply to the process “led’Off” and then call the process “ItimerProtocol’Active”. When the process “Itimer-Protocol’Active” is enabled, it will wait a certain amount of time. The time-delayed depends on the user-defined variable “TimerValue”. After the delay, it sends a notification to other processes to inform them that the time is out.

In the mCRL2 model of LED, besides user-defined actions like *SwitchOn* and *SwitchOff*, most of the actions are built in ASD actions. Those ASD actions represent behaviours relating to communications in ASD like reply events and notifications. In the modified LED model, all those actions need to be assigned with a time tag (where one time unit in the model represents 100ms in the real world). Otherwise, in a real-time mCRL2 model, an action without time tag means that it can be performed at any possible time. Therefore, a delay of 1 time unit is set between two successive actions to restrict their order of occurrence and timing is assigned to actions. As a result, the time consumption of communication in the LED model is 0.6 seconds, and to hide the lag of communication, “TimerValue” is set to 4.4 seconds so that after the button is pressed the LED could light after exact 5 seconds.

Since the precision of time is infinite, variables related to time are generally defined with real numbers. However, with real numbers, an infinite number of states will be generated in the state space which is inconvenient to observe behaviours of the model. Therefore, timing in this model is defined as a natural number, and the time to press the *SwitchOn* button is limited within 5 time units. Namely, the button can be pressed at 1, 2, 3, 4 and 5 time units.

Time in mCRL2 is absolute and not relative. Occurrences of all actions rely on their time tags. In the mCRL2 model of LED, some processes may run countless times. Therefore, it is important to record the time of the last performed action. Thus, after a process has finished, the newly started process could follow the timeline. For example, a parameter “t” is attached to the process

"ITimerProtocol'Active", in process "ItimerProtocol'Inactive" when action "ITimer'CreateTimer" takes place at time 0.5, a reply will be sent at time 0.6. Then process "ITimerProtocol'Active(0.7)" will be called, and the following action in this process will take place at "0.7+TimerValue".

Figure 3.3 indicates the state space of the modified LED model. The initial state is green, and from the initial state, there are five outgoing transitions. Those five transitions indicate different times x when the *SwitchOn* take places, where $\{x \in \mathbb{N} | 1 \leq x \leq 5\}$. The ASD state *SwitchingOn* is presented at transitions 26, 27, 28, 29 and 30. From those states, the system can choose either to wait 5 seconds or cancel the timer and go to the *Off* state.

Figure 3.4 shows a zoomed in state space of the LED model with transition labels. The action *switchOn* takes place at 0.5 seconds, 0.1 seconds later a timer with 4.3 seconds is created. When 4.3 seconds pass, the action *inevitableInternalTrigger* indicates the time is out. As a result, the LED can be turned on 5 seconds after the *SwitchOn* button is pressed.

```
proc ITimerProtocol'Active (t:Nat) =
       invoked''asomeITimer'ITimer'CancelTimer .
       writeReply(asomeITimer'ITimer'VoidReply) .
       ITimerProtocol'Inactive
     + inevitableInternalTrigger | lockQ_s(LOCK''ITimerProtocol)@(t+TimerValue) .
       pushNotification(asomeITimer'ITimerCB'Timeout)@(t+x+TimerValue) .
       unlockQ_s@(t+TimerValue+2) .
       ITimerProtocol'Inactive;

     ITimerProtocol'Inactive =
       sum x:Nat.((ITimer'CreateTimer)@x .
       writeReply(asomeITimer'ITimer'VoidReply)@(x+1) .
       ITimerProtocol'Active(x+2))
     + invoked''asomeITimer'ITimer'CancelTimer .
       writeReply(asomeITimer'ITimer'VoidReply) .
       ITimerProtocol'Inactive
     + delta;

```

Listing 3.1: Modified mCRL2 Specification of the ASD Timer

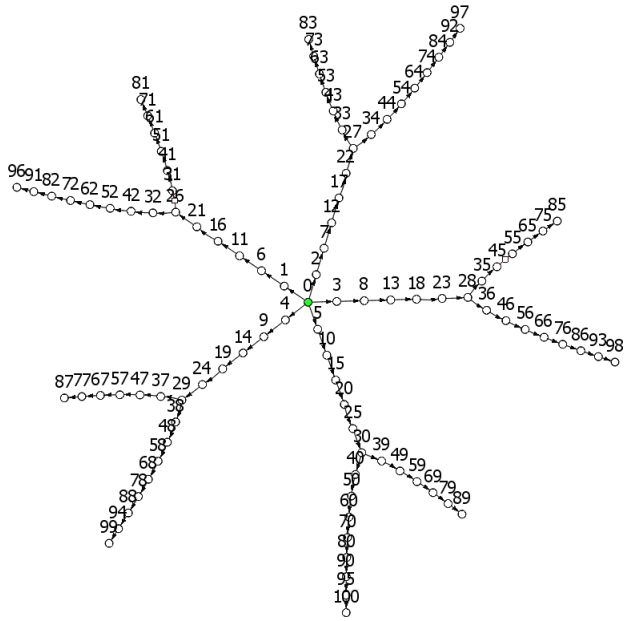


Figure 3.3: The state space of the LED system with timing

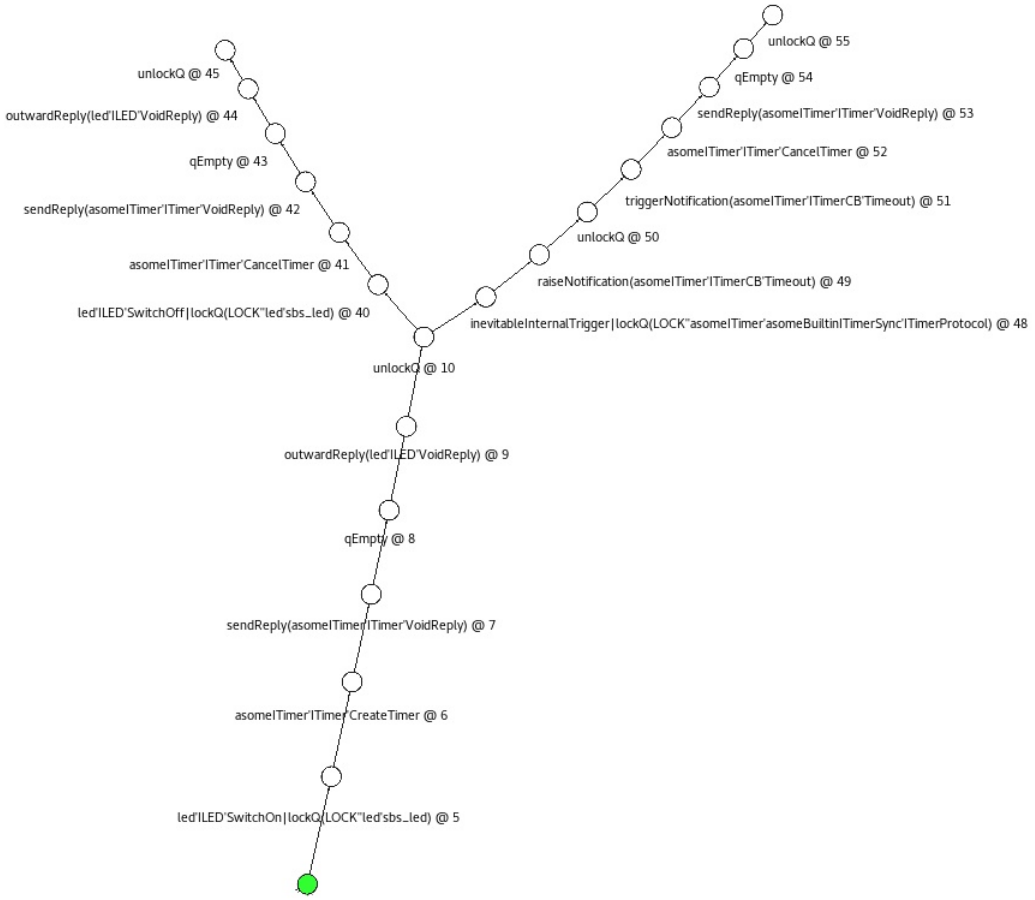


Figure 3.4: The zoomed in state space of the LED system with timing

Chapter 4

mCRL2 models based on ASML real-time requirements

In this chapter, real-time mCRL2 models are implemented with timer processes to demonstrate its suitability for ASML requirements. Real-time requirements from ASML are presented and analysed first. Then, timer processes in mCRL2 and tools to verify real-time models are introduced. Then mCRL2 models based on ASML requirements are discussed. Those models are attempted to be implemented as close as possible to real systems while keeping the models sufficiently simple such that models are understandable and can be verified by the mCRL2 tool-set.

4.1 ASML real-time Requirements

The previous chapter shows the potential of using the ASD timer to convert a real-time ASD model into a real-time mCRL2 model. However, at the front end, it is also required that the ASD Timer can be applied to the ASML real-time requirements. Therefore, the project requirements related to time are sampled from the ASML database and classified.

The database of ASML contains a significant amount of documents for all projects relates to various chip manufacturing fields. Some of those requirements do not contain real-time requirements. Furthermore, if real-time requirements are found in one document, there may also be relating real-time requirements in other documents under the same project. Therefore, simple random sampling is applied first to all the documents. If the sampled document contains real-time requirements, all the document under the same project will be inspected. As a result, 30 real-time requirements are sampled from 15 documents. The sampled ASML real-time requirements can be found in Appendix B. Based on the flexibility of the required timing, sampled real-time requirements are classified into fixed time requirements and timing variation requirements.

4.1.1 Fixed Time Requirements

In the fixed time requirements, actions have a fixed time duration and whether the requirements are met depends on a particular action that takes place within a specific period. In the sampled ASML requirements, 70% are fixed time requirements, which is widely used in ASML. Depending on when the action takes place, fixed time requirements can be further subdivided into two categories:

1. An action is expected to take place after a specific action takes place.

2. An action is expected to take place before a specific action takes place.

Below an example of a fixed time requirement extracted from ASML timing requirements relating to Item 1 is given.

Example 4.1.1. When the signal X is *False*, it will trigger a countdown time of 2 minutes. If the signal X is recovered to the status *True* within 2 minutes, the signal Y will remain *True*, otherwise the signal Y will become *False*.

In fixed time requirements, the other commonly used requirements are requirements that limit the time of processes. To achieve such requirements, actions in a process need to be tightly arranged in time. Actions need to be done before their deadlines so that the processing time does not exceed their time limits. These kinds of requirements are classified to Item 2.

As in fixed time requirements, each action has a fixed running time. ASD timer server is capable of expressing this kind of requirements. After models are translated into the ALIAS models, timer information is wholly preserved. Therefore, for fixed time requirements, only MIDS needs to be modified.

4.1.2 Timing Variation Requirements

In the timing variation requirements, the duration of some actions in a model is uncertain, and that could lead to various total running times of a process. An example of a timing variation requirement is shown below:

Example 4.1.2. The maximum timing variation of a wafer on the load robot wait position is 10s.

Two situations could lead to various total running time. The first situation is that within a program, all the actions have a fixed running time, but there exist different scenarios with a different total running time. In this situation, since all the actions have fixed running duration, the real-time model can make use of an ASD timer. The other situation is that actions do not have a fixed running time. The running time of actions is indicated with time intervals, e.g., the running time of action A varies from 1 second to 2 seconds. After action A started, it could end at any time between 1 second and 2 seconds. In this case, ASD timer cannot be used since it only accepts one input variable.

In an ASD model, timers are created with the action $Timer.ITimer.CreateTimer(\$x\$)$, where x is the duration of the timer. After the translation from ASD to ALIAS, the action $Timer.ITimer.CreateTimer(\$x\$)$ in ASD is expressed as $asomeITimer.ITimer.CreateTimer("\$x\$")$ in ALIAS. However, in ALIAS, there is not a real timer that counts time. ALIAS only extracts and saves the value from the ASD timer, and as mentioned before, no other operations are applied to a timer by MIDS either. Also, the data type of variable x is not limited to numbers.

```
1  proc ITimerProtocol'Active (t:Nat) =
2         invoked''asomeITimer'ITimer'CancelTimer .
3         writeReply(asomeITimer'ITimer'VoidReply) .
4         ITimerProtocol'Inactive +
5         sum x:Nat.(t+lowerlimit<=x&&x<=t+higherlimit)->
6         inevitableInternalTrigger | lockQ_s(LOCK''ITimerProtocol)@(x) .
7         pushNotification(asomeITimer'ITimerCB'Timeout)@(x+1) .
8         unlockQ_s@(x+2) . ITimerProtocol'Inactive;
9
10      ITimerProtocol'Inactive =
11        sum x:Nat.((ITimer'CreateTimer)@x .
12        writeReply(asomeITimer'ITimer'VoidReply)@(x+1) .
13        ITimerProtocol'Active(x+2)) +
14        invoked''asomeITimer'ITimer'CancelTimer .
15        writeReply(asomeITimer'ITimer'VoidReply) .
16        ITimerProtocol'Inactive + delta;
17
```

Listing 4.1: Modified mCRL2 Specification of the ASD Timer for timing variation requirements

To deal with the action with different running time, an additional variable is then added to the ASD action: $Timer.ITimer.CreateTimer(\$x, y\$)$, where x and y are the lower limit and higher limit of the duration. Namely, time out can take place at any time within x and y time units. In ALIAS, the expression is then translated into $asomeITimer.ITimer.CreateTimer("\$x, y\$")$.

MIDS needs to be modified to read those two values and generates a proper expression for a timer with various durations.

Listing 4.1 presents the handwritten mCRL2 specification of timer processes with varying time. The higher and lower time limits are indicated by the variable "lowerlimit" and "higherlimit".

4.2 The Simplified Timers

Two timers are created for fixed time requirements and timing variation requirements. The timer processes are used throughout all the examples in this chapter. The mCRL2 expression of those timer processes are listed in Listing 4.2 and 4.5:

```
1 proc Timer = sum x:Real.startTimer_c@x.
2                          (timeOut_c@(x+countdown)+cancelTimer_c).Timer
3
```

Listing 4.2: mCRL2 Specification of fixed timer process

In Listing 4.2, actions "startTimer_c" and "timeOut_c" communicate with other processes to indicate the start and end of the timer. The count down time from ASD Timer is defined as a constant in mCRL2 ("countdown" in the mCRL2 timer process). Thus, each mCRL2 timer process corresponds to an ASD timer service. Whenever the action "startTimer" takes place, action "timeOut" will take place after "countdown" time units, unless the action "cancelTimer" takes place.

```
1 proc UnfixedTimer =
2        sum x:Real.startTimer_Uf_c@x.
3        (sum y:Real.(x+low<=y && y<=x+high)->timeOut_Uf_c@y+cancelTimer_Uf_c).
4        UnfixedTimer;
5
```

Listing 4.3: mCRL2 Specification of unfixed timer process

Listing 4.5 indicates a timer process for timing variation processes. This timer process contains two inputs variables, "low" and "high" are the lower time bound and higher time bound of the timer. Namely after the action "startTimer_Uf" takes place, the action "timeOut_Uf" can take place at any time after "low" time units and before "high" time units. With this unfixed timer, it is possible to implement a model with flexible action duration.

4.3 Verification tools

In the mCRL2 tool-set, verification of real-time models is achieved by *pbessymbolicbisim* and *lpssymbolicbisim*. Where *pbessymbolicbisim* aims at solving parameterised Boolean equation systems with an infinite underlying Boolean equation system, *lpssymbolicbisim* aims at generating the reachable part of the bisimulation quotient of systems with an infinite state space ([12]). Those are two important tools to verify real-time properties with real-time mCRL2 models. Currently, those two tools are still in the experimental stage and results are not guaranteed to be generated with complex mCRL2 models by using those tools.

With ASD built-in communication mechanisms, the transformed ASD models are always too complex for *pbessymbolicbisim* and *lpssymbolicbisim*. Therefore, the concepts from the previous chapter are not directly applied to MIDS. Instead, for each classified requirement, an mCRL2 model is made with a simplified timer process where the mechanism is the same as the ASD timer, and ASD communication functions are eliminated. The purpose of implementing such models is to prove that it is feasible and reliable to apply concepts from the last section to MIDS so that real-time models can be implemented with ASD and properties can be verified by mCRL2 tool-set.

4.4 The Signal indicator model

The signal indicator example is a model implemented based on Example 4.1.1. In the signal indicator model, there exist two main signals: signal X and signal Y, two sub-signals: signal A and signal B. In the initial state, all the signals are true. The state of signal X relies on signal A and signal B, which means if one of those two signals becomes false, signal X will turn into false. If the signal X is true, it is possible that signal A becomes false at any time. As soon as the system notices that signal A is false, it will start the recovery process.

The process of recovering signal A takes 100 seconds. After the recovery process is done, signal A returns to true. However, the failure of signal A may cause signal B also to become false, which means signal B can turn into false during the recovery process of signal A; then the recovery process of signal B will start. The recovery process of signal B takes 80 seconds. Signal A and signal B can be recovered simultaneously. If signal A and signal B cannot be recovered within 120 seconds, signal Y will become false, and the *reset* action needs to be performed to fix signal X to true.

In this model, the recovery processes have a fixed duration. Therefore, the model could be modelled with the fixed timer. Based on this requirement, an mCRL2 model is implemented. Actions are listed in the table 4.1.

Actions	Description
startTimer	Start countdown of 120 seconds.
startTimerA	Start countdown of 100 seconds.
startTimerB	Start countdown of 80 seconds.
cancelTimer	Cancel the 120 s timer before time out
cancelTimerA	Cancel the 100 s timer before time out
cancelTimerB	Cancel the 80 s timer before time out
timeOut	The 120 s timer finished countdown
timeOutA	The 100 s timer finished countdown
timeOutB	The 80 s timer finished countdown
signalY(b)	Signal Y changes to Boolean b
signalA(b)	Signal A changes to Boolean b
signalB(b)	Signal B changes to Boolean b
recoverDone	Signal X changes to True
reset	Reset the system

Table 4.1: List of actions in the Example 3.2.1

In this example, three timers processes are defined: *Timer*, *TimerA* and *TimerB*. Process *Timer* has a countdown value of 120 seconds, corresponds to the deadline of signal Y to become false. Process *TimerA* and *TimerB* have countdown values of 100 and 80 seconds respectively to indicate the recovery processes of signal A and signal B.

As soon as signal A becomes false, the timer *Timer* and *TimerA* will be triggered, and then the recovery process of signal A will start. Timer *TimerB* is triggered when signal B becomes false, and the system will start recovering *signal B*.

In this example, the state space is used to analyse the behaviours of the model. The state space of Example 3.2.1 is presented in Figure 4.1, where the initial state is marked with green. The state space is generated by using three tools: *mcrl22lps*, *lpsuntime*, *lpssymbolicbisim* and *ltsgraph*. Where *mcrl22lps* transfers mCRL2 specification into a linear process, *lpsuntime* eliminates time from a linear process specification, so that state space can be generated with *lpssymbolicbisim*. The graph of state space is visualised by *ltsgraph*. In this transformation, although time is eliminated by *lpsuntime*, behaviours of the model would not be changed. Commands used to generate the state space are listed in Listing 4.4. The mCRL2 specification is listed in Appendix C.1.

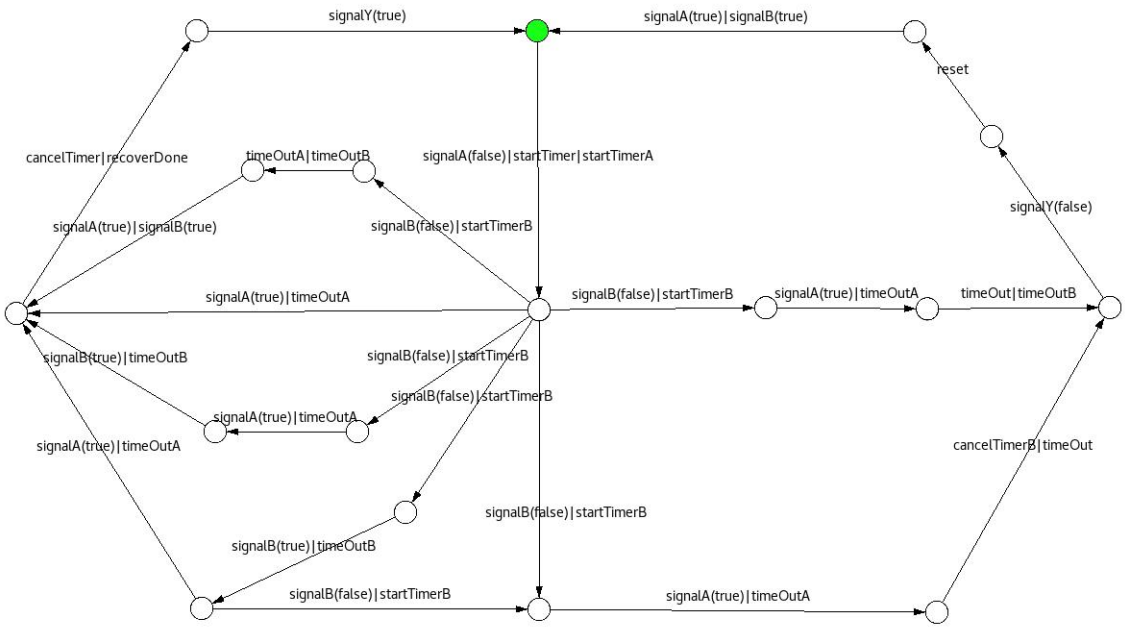


Figure 4.1: The state space of the Signal indicator model

```
$ mcrl22lps signalIndicator.mcrl2 signalIndicator.lps -Tfwno
$ lpsuntime signalIndicator.lps signalIndicator.lps
$ lpssymbolicbisim signalIndicator.lps
$ ltsgraph out.lts
```

Listing 4.4: Commands used to generate the state space of the signal indicator model

By analysing the state space, the first property can be checked is that the model is deadlock free. Furthermore, after action *signalA(false)* takes place, there exists 6 transitions, and within those 6 transitions there are 5 transitions with the same transition label *signalB(false)*. Even though their transition labels are the same, each of the transition indicates different timing that *signalB(false)* takes place. Figure 4.2 presents different timelines after the action *signalA(false)* takes place. In the figure, the path that only signal A becomes false is not presented; in this situation, signal Y will not become false.

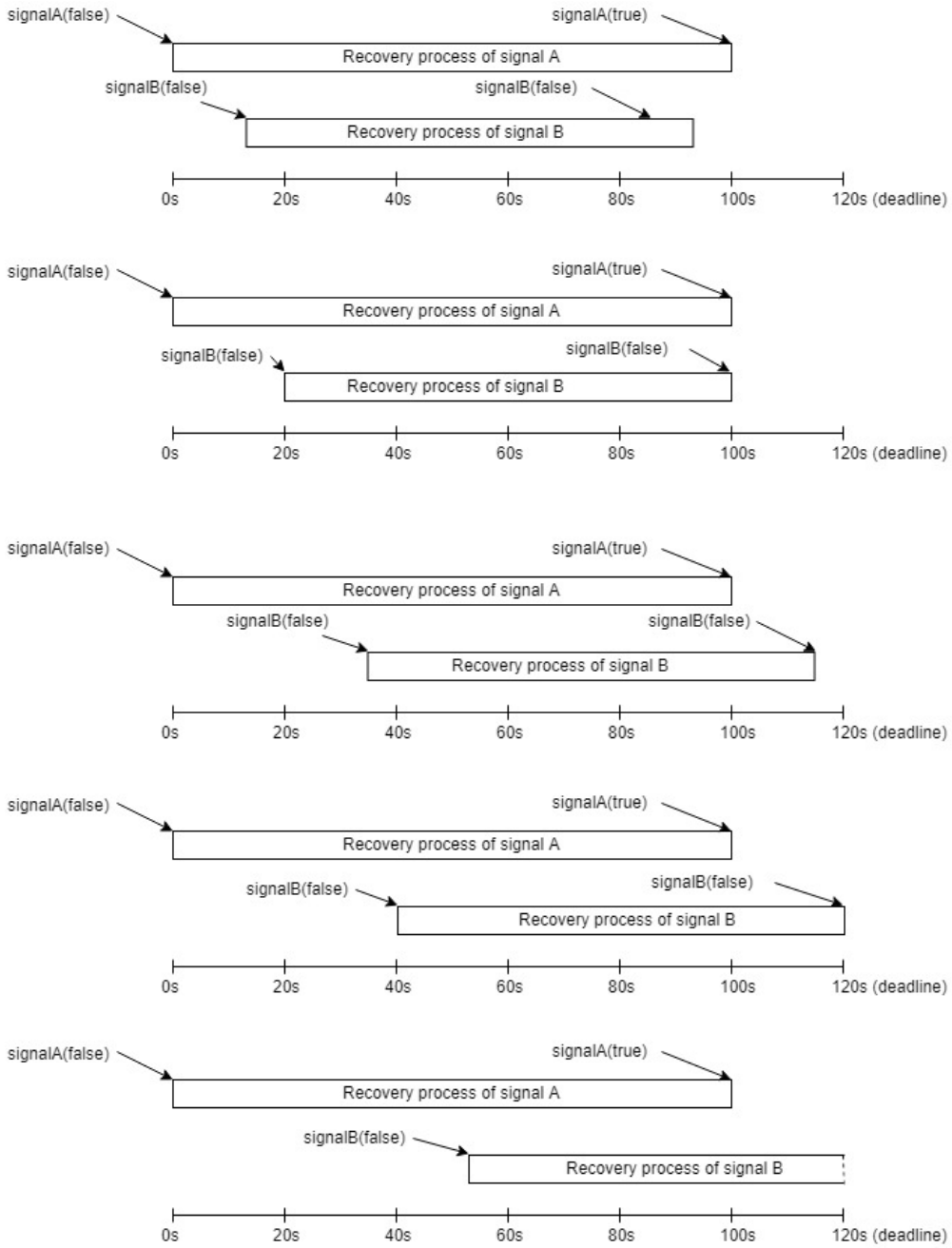


Figure 4.2: Five different situations after the action *signalA(false)* takes place

4.5 Multi-processors model

The model of multi-processors corresponds to the fixed time requirements that tasks are restricted to be finished before a firm deadline. In this example, an application is limited to be finished within 35 time units; this application contains three tasks t1, t2 and t3. The time required to complete these three tasks is 10 time units, 20 time units and 10 time units. The task t2 and t3 can start after task t1 are done. These tasks can be assigned to 2 processors: processor 1 and processor 2. The processing speed of those two processors is the same. Communication between processors also takes time, communication time from task t1 to t2 is 5 second, from t1 to t3 is 10

seconds. The architecture of this example is shown in Figure 4.3.

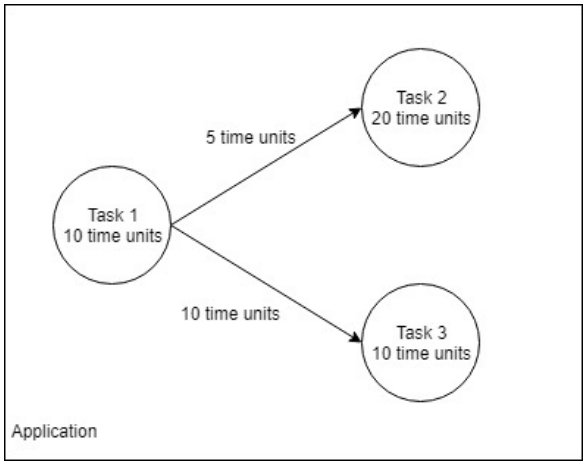


Figure 4.3: The architecture of the application in multi-processors example

An mCRL2 model is built to prove that with such time configurations of three tasks, there exists a scheme that the application can be finished within 35 time units. In the mCRL2 model, a process *Controller* assign tasks to two processors. This process contains 6 parameters: t1, t2, t3, p1, p2 and t, where t1, t2 and t3 indicate the state of task 1, task 2 and task3.

For parameters t1, t2 and t3, each of them contains 7 states to indicate whether the task is being processed, communicating, have been finished on processor 1 or processor 2. Variable p1 and p2 indicate the current processing task on processor 1 and processor2. Variable t records time of last performed action. Each choice the process controller made is based on the state of tasks. Each task is assigned with two actions; an action indicates the start of the action and the other action indicates the task is done. Actions used in the example are listed in the table 4.2. The mCRL2 specification of the model can be found in Appendix C.2.

Actions	Description
startTimer1(x),startTimer2(x),startTimer3(x)	Start a timer with countdown of x time units
timeOut1,timeOut2,timeOut3	The timer has finished countdown
StartTask1,StartTask2,StartTask3	Start the task t1, t2 and t3
Task1Done,Task2Done,Task3Done	Task t1, t2 and t3 is done
commuT1_T2	Start communication between task t1 and t2
commuT1_T3	Start communication between task t1 and t3
programDone	All the tasks are done

Table 4.2: List of actions in the Example 3.2.2

Although it is a simple application which only contains three tasks, the model is still relatively complex for the mCRL2 tool-set. Therefore, two adjustments are made to simplify the model. Because two processors are the same, task t1 is forced to execute on the processor 1. Thus, states of the model are decreased, and that lowers the difficulties of verifying the model. Furthermore, different from the previous example that each action has its timer; in this model, a timer can be shared by different actions. This is because the processing time and memory usage increases exponentially with the number of parallel processes exists in a real-time mCRL2 model. The modified timer process is listed in Listing 4.5.

```
proc Timer = sum x,c:Real.startTimer_c(c)@x.
                          (timeOut_c@(x+c)+cancelTimer_c).Timer

```

Listing 4.5: mCRL2 Specification of fixed timer process with adjustment

With the expression in Listing 4.5, the count down value is carried by the action "startTimer_c". For example, when task t1 starts on the processor 1, it will start a timer with countdown of 10 time units, this behaviour can be presented as startTask1(1)|startTimer1(10) in the mCRL2 expression. Other actions cannot access this time until task1Done(1)|timeOut1 (task1 done on the processor 1) takes place. The reason for this adjustment is to reduce the number of parallel processes and increase the speed of processing since a large number of parallel processes can cause slow processing when verifying the model.

Although using the timer process listed can lead to a faster processing speed with the mCRL2 tool-set, it could also bring disadvantages. Since the timer can be shared, if an action has called the timer process, other action cannot use this time process until action timeOut takes place. Therefore, if the timer process is not assigned correctly, the behaviour of the system could be changed. The difficulty of modelling will increase by applying this shareable timer. Actions must be arranged comprehensively so that no conflicts are competing for timers. This adjustment is difficult to be applied on MIDS because it requires MIDS to recognise actions contain time constraints that could run simultaneously and then assign timer processes to those actions.

In this model, there are three timers used. Timer "Timer 1" is responsible for timing on core 1, timer "Timer 2" is responsible for timing on core 2. Time spent on communications is handed by timer "Timer3". Comparing with arranging a timer for each task and communications, the adjusted timer reduced the number of parallel processes by two.

The state space of this example is presented in Figure 4.4. Commands used to generate the state space are listed in Listing 4.7 From the state space it can be seen the model is deadlock and livelock free since every state contains a path to the initial state. However, since the state space is relatively complex, it is difficult to observe behaviours of the system directly. Therefore, to verify such properties that after StartTask1 takes place, it is possible that programDone takes place within 35 time units, the modal formula expression listed in Listing 4.6 can be used. Commands used to verify the modal formula are listed in 4.8.

```
[true*]forall t:Real. [startTask1(1)|startTimer1(10)@t]<true*>exists u:Real.
(val(u<=(t+35))&& <programDone@u>true)

```

Listing 4.6: Modal formula to verify that after StartTask1 takes place it is possible programDone takes place within 35 time units

```
$ mcrl22lps multiProcessor.mcrl2 multiProcessor.lps -Tfwno
$ lpsuntime multiProcessor.lps multiProcessor.lps
$ lpssymbolicbisim multiProcessor.lps
$ ltsgraph out.lts

```

Listing 4.7: Commands used to generate the state space of the multiprocessor model

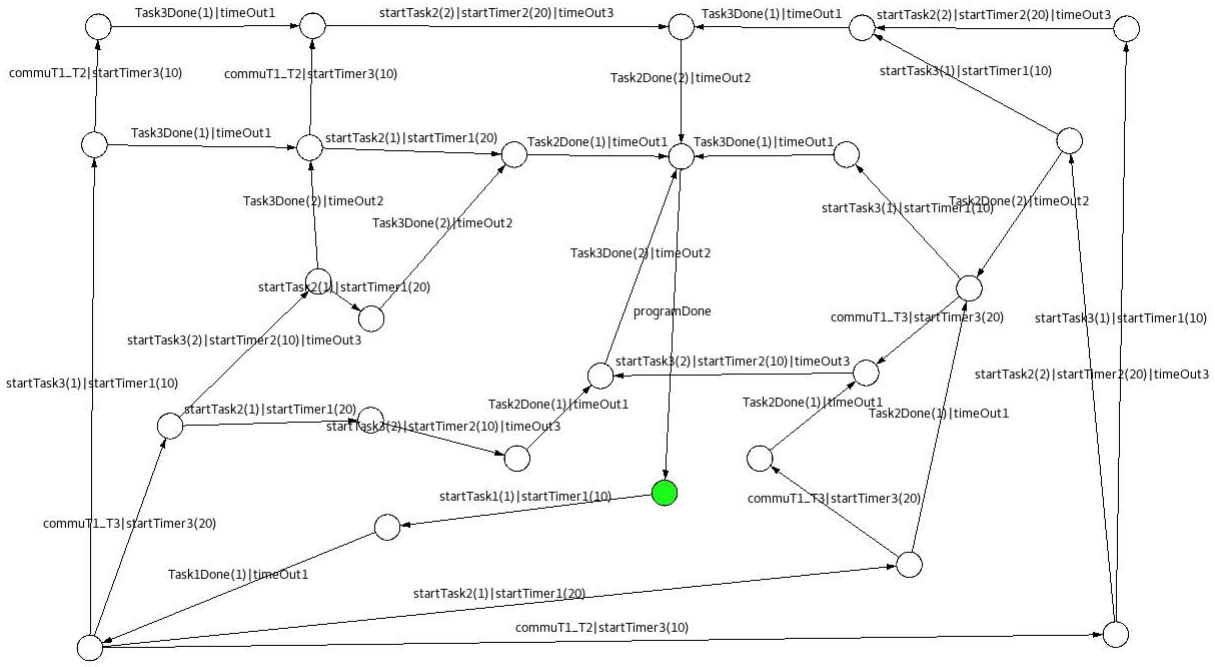


Figure 4.4: The state space of the multi-processors example

```
$ mcrl22lps multiProcessor.mcrl2 multiProcessor.lps -Tfwno
$ lps2pbes multiProcessor.lps -fmf.mcf multiProcessor.pbes -t
$ pbessymbolicbisim multiProcessor.pbes

```

Listing 4.8: Commands used to verify the modal formula of the multiprocessor model

Regrettably, for this example, the current mCRL2 tool-set could only verify there exists a path that the application is finished within 35 time units, it cannot provide more information such like the optimal task allocation. That is because the tool pbessymbolicbisim can only output the correctness of a property.

4.6 Data flow model

In the sampled ASML requirements, there is also one requirement which relates to the throughput. In a lithography machine, throughput is an important parameter that reflects the production efficiency. Throughput indicates a total number of units the process can produce divided by the processing time. In this section, a data flow mCRL2 model is implemented to demonstrate how properties relate to throughput can be verified.

The model is implemented based on the single rate data flow graph ([13, Chapter 2]) in Figure 4.5. In the single rate data graph, tasks are expressed by cycles. There are three tasks in total: T1, T2 and T3. The processing time of each task is also mentioned in the graph (number after task name). In the data flow graph, whether a task can start firing depend on the number of tokens on the input edges. The token is presented as black dots near edges. Each time the task

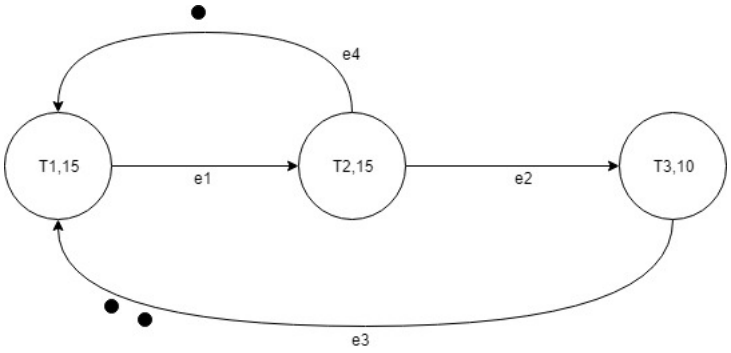


Figure 4.5: The data flow graph of the data flow example

fires, it consumes one token from every input edge. After a task is done, it will generate a new token to all of its output edges.

Similar to the previous multi-processor model, in the mCRL2 model, tasks are controlled by a process called controller. This process contains 8 inputs, where e1, e2, e3 and e4 are defined as natural numbers to indicate the number of tokens on each edge. Variable t1, t2 and t3 are defined as Boolean to indicate the state of the task, where False means the task is not being processed and True means the task has started and not finished yet. A real number t is used to record the time of the last performed action.

Each task involves two actions; an action indicates the start of the task and the other shows the task is finished. The complete list of actions used in this model is listed in Listing 4.3. The mCRL2 specification of the model can be found in Appendix C.3.

Actions	Description
startTimer1(x),startTimer2(x)	Start a timer with countdown of x time units
timeOut1,timeOut2	The timer has finished countdown
StartTask1,StartTask2,StartTask3	Start the task t1, t2 and t3
Task1Done,Task2Done,Task3Done	Task t1, t2 and t3 is done

Table 4.3: List of actions in the data flow model

In this model, the parallelism of the tasks is 2, which means the total processing time is the lowest when tasks are running on two processors. To simplify the model, the shareable timer processes are used to reduce the number of parallel processes. In this model, the number of timer processes used is equivalent to the number of processors involved, and the model could behave differently when a different number of timer processes are added. Therefore, two mCRL2 models with one timer process and two timer processes are built.

The state space of the model with one timer process involved is presented in Figure 4.6. As can be seen from the figure, the shortest path from the initial state to the occurrence of action Task3Done is to process Task 1, Task 2 and Task 3 in order. The total processing time of the shortest path is 43 time units (processing time of each task plus 1 time unit delay between tasks). With 43 time units total processing time, the throughput of the system is $\frac{1}{43}$ fires per time unit.

The state space of the model with two timer processes involved is presented in Figure 4.7. As the state space is more complex than the model with one timer process. It is difficult to determine the shortest path directly from the graph. With the method introduced in [13, Chapter 2], the maximum throughput can be calculated with *loop bound*. Loop-bound in single rate graph can be calculated as the round-trip time consumption in a given loop of a data flow graph, divided by the number of tokens in that loop. The maximum throughput is the highest loop bound for any loop in a given data flow graph.

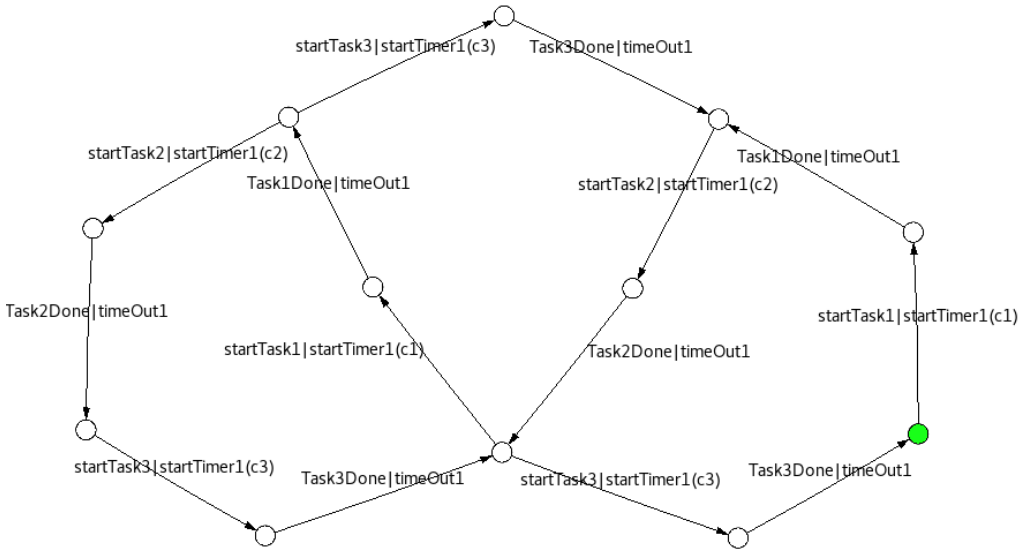


Figure 4.6: The state space of throughput model with one timer process

In this example, there exist two loops: T1T2 and T1T2T3. The loop T1T2 has a running time of 32 time units (include 2 time units delay) with 1 token, and the loop T1T2T3 consumes 43 time units with two tokens. As a result, the loop bound of those two loops are $\frac{1}{32}$ and $\frac{2}{43}$ fires per time unit. The maximum throughput is then $\frac{1}{32}$ fires per time unit. Figure 4.8 indicates the allocation of tasks that maximum throughput can be achieved.

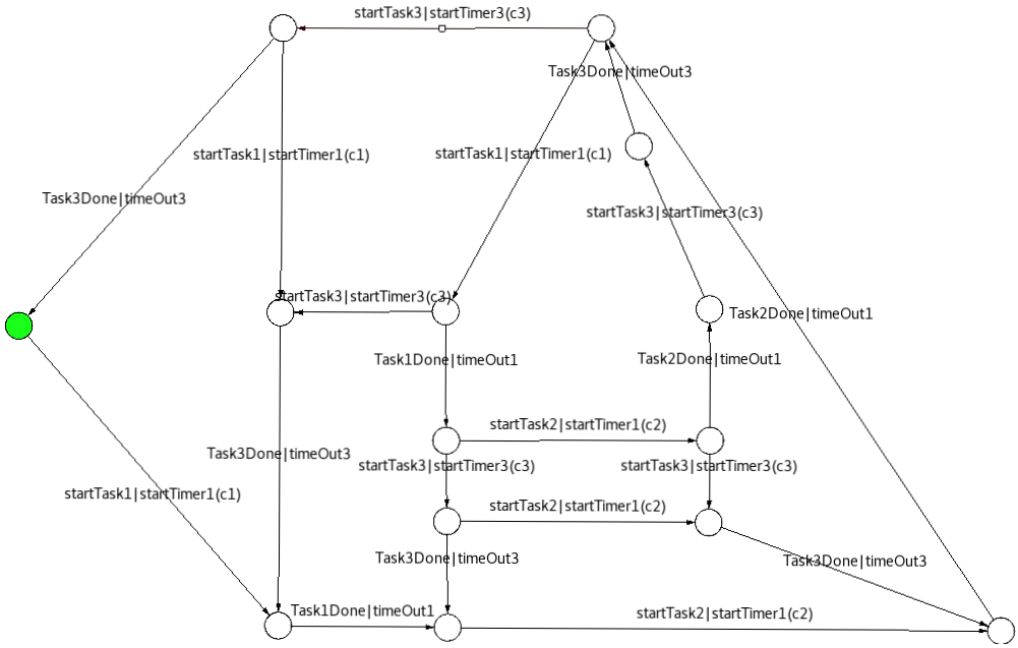


Figure 4.7: The state space of throughput model with two timer processes

To prove that the throughput is higher or equal than $\frac{1}{32}$ fires per time unit, the modal formula in Listing 4.9 can be used. The modal formula expresses that after action startTask1 takes place, it is possible that Task3Done takes place within 32 time units. Commands used to generate the state space and verify modal formula is listed in Listing 4.10 and 4.11.

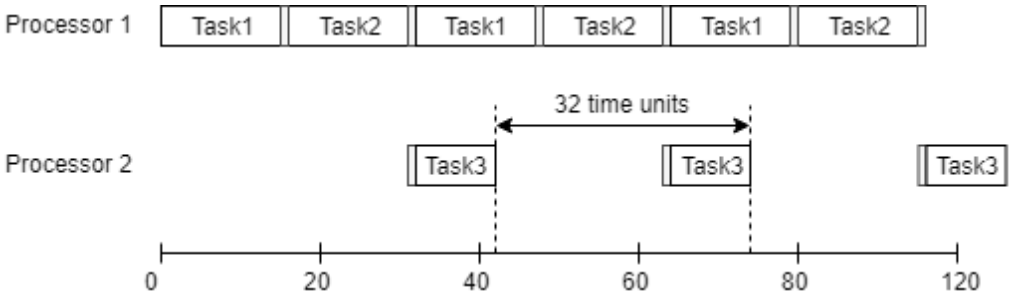


Figure 4.8: Task allocation of maximum throughput

```
[true*]forall t:Real. [startTask1|startTimer1(c1)@t]<true*>exists u:Real.
(val(u<=(t+32))&& <Task3Done|timeOut3@u>true)

```

Listing 4.9: Modal formula to verify throughput is higher or equal than $\frac{1}{32}$ time units

```
$ mcrl22lps multiProcessor.mcrl2 multiProcessor.lps -Tfwno
$ lpsuntime multiProcessor.lps multiProcessor.lps
$ lpssymbolicbisim multiProcessor.lps
$ ltsgraph out.lts

```

Listing 4.10: Commands used to generate the state space of the data flow model

```
$ mcrl22lps dataFlow.mcrl2 dataFlow.lps -Tfwno
$ lps2pbes dataFlow.lps -fmf.mcf dataFlow.pbes -t
$ pbessymbolicbisim dataFlow.pbes

```

Listing 4.11: Commands used to verify the modal formula of the data flow model

4.7 Wafer stepper model

In this section, an mCRL2 model is built based on Example 4.1.2 that models wafers which need to wait on the load robot, and the waiting time varies. It is required that the waiting time of each wafer in the wait position is less than 10 seconds. It is challenging to model actions only with the fixed-timer process since it only deals with a fixed time duration. Although multiple fixed-timer processes can be set, it can only be applied when all the possible last duration of action is already known. Since the purpose of this model is to demonstrate the unfixed timer mCRL2 process, the timer process from Listing 4.5 is used to implement this example.

In the mCRL2 model, there are two kinds of wafers (wafer A and wafer B); each kind of wafer needs a different time duration to be processed. The processing time of wafer A varies from 150 seconds to 170 seconds and the processing time of wafer B varies from 200 to 220 seconds. Before a wafer is processed, it will be delivered to a wait position first. In the wait position, the wafer will wait until the previous wafer is processed.

A schematic view of this process is listed in Figure 4.9. This schematic view indicates an example of how the wafer stepper deal with two consecutive input wafers: wafer A and wafer B. In the initial state, both the wait position and processing position are empty. Then, wafer A is delivered to the wait position. Since the processing position is empty, the wafer A is delivered to the processing position immediately after it reached the wait position. As long as the wait position is empty, a new wafer can be delivered to the wait position (in this case, wafer B is delivered). After the processing of wafer A is done, wafer B will be delivered to the processing position.

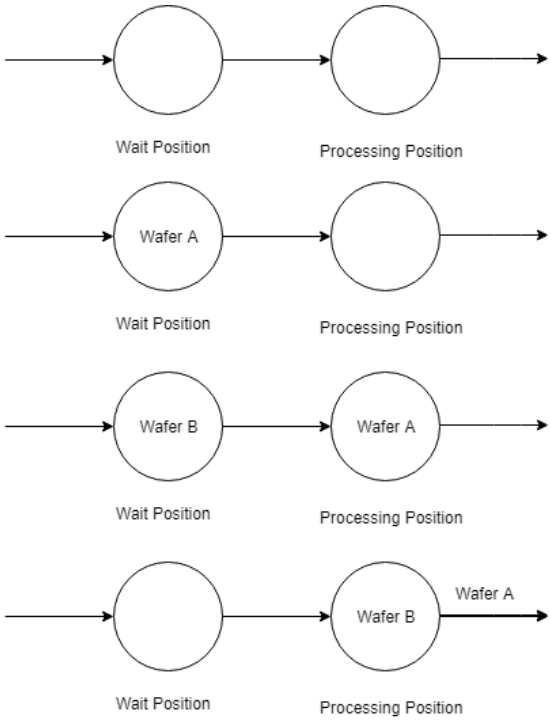


Figure 4.9: Schematic overview of the wafer stepper model

Before the model is implemented, the strategy of wafer delivering is determined first. To keep the waiting time as short as possible, only when the previous wafer is about to be processed, a new wafer will be delivered to the wait position. Thus, the waiting time for each wafer is less than 10 seconds.

The strategy is presented in Figure 4.10. There are three situations based on the wafer type being processed. If the wafer being processed is type A, then a new wafer should be delivered to the wait position at 150 seconds after the start of the processing of wafer A. If wafer B is being processed, the new wafer should be delivered to the wait position at 200 seconds after the start of the processing of wafer B. There is also a situation not shown in the figure that the processing is done before the countdown. In this case, the timer would be, and a new wafer will be delivered to the wait position immediately.

Actions used in this example are listed in Listing 4.4. The mCRL2 specification of the model can be found in Appendix C.4. In this model, there are three processes; a timer process "TimerA" responsible for processing time of wafer A and wafer B, the other timer process "Timer1" is modelled to arrange the delivery of next wafer. A process Controller controls the transportation of wafers on the wafer stepper.

The state space of the wafer stepper model is shown in Figure 4.11. Commands used to generate this state space are shown in Listing 4.13. As can be seen, for both types of wafers, after the action startProcessing, there exist two choices: the timer Timer1 finishes its countdown, and a new wafer is delivered to the wait position while a wafer is still being processed. The other choice is that the timer TimerA finished its countdown before the timer Timer1. Then a new wafer is delivered to the wait position, and the timer Timer1 is cancelled. By checking the state space, we can determine that the model is consistent with the strategy. However, it is still not clear if the waiting time of each wafer is less than 10 seconds. Therefore, modal formulas are used to check this property.

The property can be checked separately for wafer type A and wafer type B with the modal formulas listed in Listing 4.12. Those modal formulas declare that after the action waferAReceived or waferBReceived take place, action waferADelivered or waferBdelivered must take place within

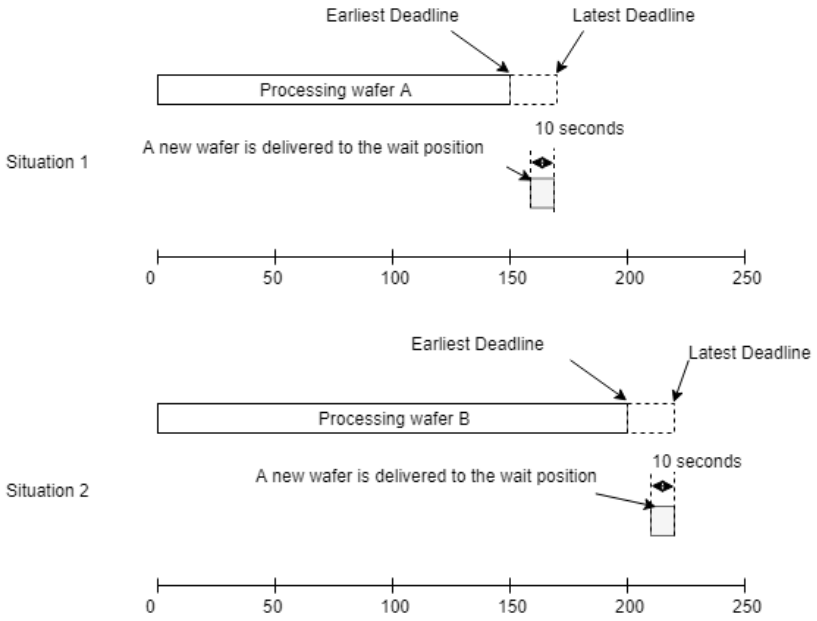


Figure 4.10: Wafer deliver strategy of the wafer stepper model

Actions	Description
startTimer_A, startTimer1	Start a timer
timeOut_A, timeOut1	The timer has finished its countdown
cancelTimer1	Cancel the countdown of timer1
waferAReceived	In the wait position, a wafer with wafer type A is received
waferBReceived	In the wait position, a wafer with wafer type B is received
waferADelivered	A wafer A is delivered from wait position to processing position
waferBDelivered	A wafer B is delivered from wait position to processing position
loadReady	Processing position is ready to load a new wafer
startProcessing	Start processing a wafer
processingDone	Processing is done

Table 4.4: List of actions in the wafer stepper model

10 seconds. As a result, those modal formulas hold for the wafer stepper model. Commands used are listed in Listing 4.14.

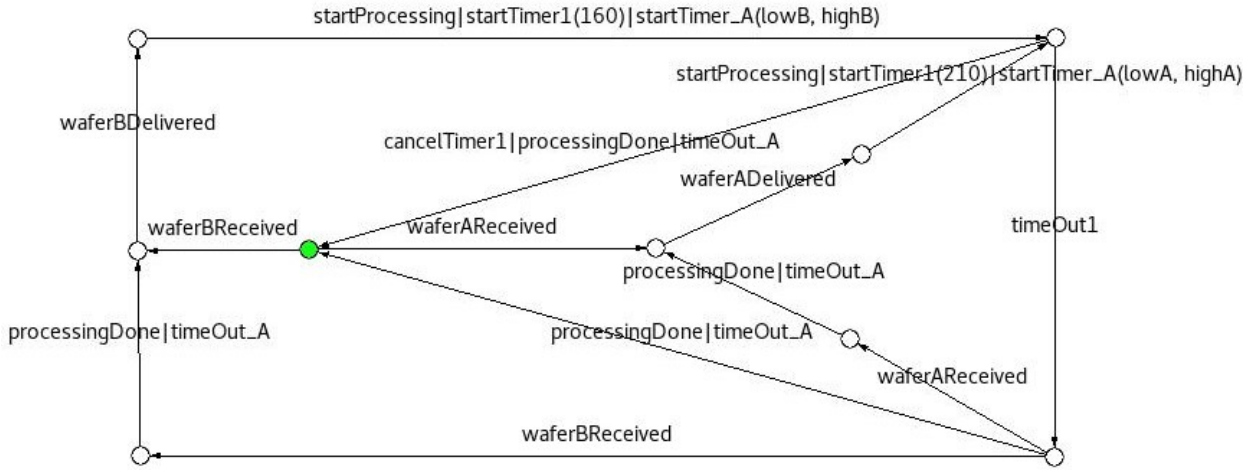


Figure 4.11: The state space of the wafer stepper model

```
[true*]forall x:Real.[waferAReceived@x]mu X.
([forall y:Real.!(val(y<=x+10)&&waferADelivered@y)]X&&<true>true)

[true*]forall x:Real.[waferBReceived@x]mu X.
([forall y:Real.!(val(y<=x+10)&&waferBDelivered@y)]X&&<true>true)

```

Listing 4.12: Modal formulas to verify the wafer stepper model

```
$ mcrl22lps waferstepper.mcrl2 waferstepper.lps -Tfwno
$ lpsuntime waferstepper.lps waferstepper.lps
$ lpssymbolicbisim waferstepper.lps
$ ltsgraph out.lts

```

Listing 4.13: Commands used to generate the state space of the wafer stepper model

```
$ mcrl22lps waferstepper.mcrl2 waferstepper.lps -Tfwno
$ lps2pbes waferstepper.lps -fmf.mcf waferstepper.pbes -t
$ pbessymbolicbisim waferstepper.pbes

```

Listing 4.14: Commands used to verify the modal formula of the waferstepper model

4.8 Discussion

With four mCRL2 real-time models implemented with timer processes, suitability of applying timer processes to ASML requirements is proved. mCRL2 timer processes can handle all existing logical relationships about time from gathered ASML requirements. However, adding timer process features to the model conversion system at this stage is not appropriate.

Models presented in this chapter are hand-written, and those models are highly optimised to let the mCRL2 tool-set generate state space and verify properties, e.g., communication functions of ASD models are not involved. Even if time features are added to the modal conversion system, translated mCRL2 models cannot be verified since the complexity of ASD communication mechanism itself has already exceeded the capabilities of the mCRL2 real-time tools.

Although at this stage it is unrealistic to add time features to the model conversion system, future modifications to MIDS are proposed. In the future, when the mCRL2 toolset is capable of dealing with complex real-time models efficiently, the following modifications can be applied to the model conversion system:

1. For fixed time requirements, MIDS should read the value contained in the timer: Each ASD timer contains a value t that indicates the delay time of that timer. MIDS should be able to read the value t from ALIAS. In the generated mCRL2 model, it needs to define a variable and assign the value t to it. In this way, timer processes can delay a proper amount of time.

2. For time variation requirements, it is possible that the ASD timer contains two values that indicate the lower and higher limit of a time delay. Therefore, when there are two values separated by a comma contained in an ASD timer, MIDS should be able to recognise those two values and transfer the timer into an mCRL2 timer process that can generate a delay between the lower and higher time limits.

3. Complete timer processes translation: As mentioned previously, with the model directly transformed from ALIAS and MIDS, timer processes are not complete. Replies and notifications after the timer is created and cancelled are missing. After a timer is created, a reply action should take place, and after a timeout, a notification should be sent to corresponding processes.

4. Apply time tags to actions: In order to constrain the order of actions and avoid deadlocks, tags need to be assigned to actions relate to time requirements.

5. Add a parameter value to processes: A parameter should be attached to processes to indicate the time at which the last action took place.

6. Define multi-actions in the mCRL2 model: In real-time mCRL2 models with parallel processes, it is possible that actions from different processes take place at the same time. In mCRL2, actions take place together are called multi-actions, and those multi-actions need to be declared in the mCRL2 model. Otherwise, no actions will take place and deadlock will be caused. Therefore, MIDS needs to sort actions used by different processes, and then list all the combinations as multi-actions.

Chapter 5

Related Work

In this project, real-time modelling is achieved by the mCRL2 toolsets. However, there are also other model checkers capable of analysing timed automata.

Kronos Kronos [14, p. 161] is a model checker using Timed Computation Tree Logic (TCTL) to solve timed automatons. TCTL is a branching-time logic using a tree-like structure to present events in future (the tool lpssymbolicbisim has a similar feature). One of Kronos's advantage is that it allows verifying of liveness properties and not restricted to reachability properties. However, it contains no graphical nor simulation modes to visualise the model more intuitively.

UPPAAL UPPAAL [14, p. 153] is a model checker that uses simplified TCTL comprised of path formulae and state formulae, where individual states are described by state formula, and path formulae quantify over path or traces. In Uppaal, a finitestate symbolic semantics of networks is used as the model-checking procedure for real-time models.

The IF toolset The IF toolset [15] is an environment for modelling and validation of heterogeneous real-time systems. It allows structured automata-based system representations. The IF notation can support real-time primitives and extensions of high-level modelling languages such as SDL and UML.

DREAM DREAM [16] stands for the Distributed Real-time Embedded Analysis Method, it is an open-source tool to verify and analyse distributed real-time and embedded systems. It focuses on the practical application of formal analysis methods. The model checking method of DREAM is realised by utilising UPPAAL or Verimag IF model checkers.

TAPAAL TAPAAL [17] is a modelling, simulation and verification tool for Timed-Arc Petri nets. Where Timed-Arc Petri Net is a time extension of the classical Petri net model. The time extension combines the concept of tokens in Petri Net with time, which means each token has its age. Arcs from places to transitions are labelled by time intervals that restrict the age of tokens that can be used in order to fire the respective transition. TAPAAL models can be translated into UPPAAL and use the UPPAAL verification engine to verify properties.

Chapter 6

Conclusions

In this work, the objective is to research the ASD timer service and to propose a way to express ASD timers in mCRL2. To achieve the goal, the ASD timer service is investigated, ASML requirements are gathered and analysed, and mCRl2 models are extended with ASD timers to illustrate the suitability of the ASD timer when coming across ASML real-time requirements.

A LED real-time model is implemented with ASD and then translated into an mCRL2 model. By analysing the mCRL2 model, problems that are related to real-time models in the modal conversion system are located: The ASD timer service is not translated into mCRL2 processes completely, reply events and notifications in ASD timer service are not translated.

Next, ASML requirements are gathered and categorised. Based on the flexibility of the required timing in those requirements, those ASML real-time requirements are classified into fixed time requirements and timing variation requirements. The ASD timer can deal with fixed time requirements. For the timing variation requirements, the ASD timer is limited by its number of inputs. To solve this problem, a modification to MIDS is proposed that when the input data in the ASD timer service are two real numbers separated by a comma, MIDS should recognise those two numbers. Thus, by using a number as a lower time limit while the other as higher time limit, the ASD timer can be used to express any time in an interval.

To verify the reliability of proposed modifications to MIDS, mCRL2 real-time models based on the gathered requirements are implemented and verified. Timing in those models is expressed with the same format as the translated ASD timer service.

As a result, four mCRL2 models correspond to different types of ASML real-time requirements are implemented. Some common modal formulas that can be generally applied on each type of real-time requirement are proposed and verified.

During the implementation, compromises are made. In order to reduce the number of parallel processes, timer processes can be invoked by multiple processes. Furthermore, in order to keep the models simple that tools could generate results, no ASD communication mechanisms are involved in the implemented mCRL2 models. Although this work proves that the ASD timer can be translated into the mCRL2 process and can be used to express real-time models adequately, the scalability of an ASD timer is limited by the real-time mCRL2 tools.

Future work ASD models contain multiple components. In this project, requirements are assumed to be related to a single component. However, in real situations, it is possible that a real-time property relating to multiple components needs to be verified. To solve this problem, the model conversion system should be modified to be able to extract all behaviours correspond to the property from related components. Then, the model conversion system transfer those behaviours into an mCRL2 model, and the property can be verified using μ-calculus. Another possible solution is to split a property into sub-properties; each of the corresponding components is assigned with a sub-property. If each sub-property holds on the assigned component, the property holds.

Chapter 7

Bibliography

[1] J.F. Groote, A.A.H. Osaiweran, and J.H. Wesselius. Analyzing the effects of formal methods on the development of industrial control software. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011, Williamsburg VA, USA, September 25-30, 2011)*, pages 467–472, United States, 2011. Institute of Electrical and Electronics Engineers (IEEE). 1

[2] G. H. Broadfoot. Analytical software design case: MagLev Stage software design. *Embedded Systems Conference ESS2005 Incorporating the IEE FPGA Developers Forum*, 2005. 1, 3

[3] G. H. Broadfoot. ASD Case Notes: Costs and Benefits of Applying Formal Methods to Industrial Control Software. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, pages 548–551, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 3

[4] W. Fokkink. Algebra of Communicating Processes. *Introduction to Process Algebra Texts in Theoretical Computer Science. An EATCS Series*, pages 17–30, 2000. 7

[5] J. F. Groote and M. R. Mousavi. *Modeling and analysis of communicating systems*. The MIT Press, 2014. 7

[6] J. F. Groote, A. Ponse, and Y. S. Usenko. 0. *The Journal of Logic and Algebraic Programming*, 48(1-2):39–70, 2001. 8

[7] J. F. Groote and T.A.C. Willemse. Parameterised Boolean Equation Systems. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 308–324, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 8

[8] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An overview of the mcrl2 toolset and its recent advances. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 8

[9] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. 8

[10] J. F. Groote and M. Keinänen. Solving disjunctive/conjunctive boolean equation systems with alternating fixed points. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 436–450, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 8

[11] E. R. Sahagun. Translation of ASD Single-threaded Execution Model to EFSM. Master's thesis, TU/e, 2014. 8

[12] T. Neele, T. A. C. Willemse, and J. F. Groote. Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotientin. In K. Bae and P. C. Ölveczky, editors, *Formal Aspects of Component Software*, pages 216–236, Cham, 2018. Springer International Publishing. 17

[13] P. R. Schaumont. *A practical introduction to hardware/software codesign.* Springer, 2010. 23, 24

[14] B. Bérard. *Systems and software verification: model-checking techniques and tools.* Springer, 2011. 31

[15] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The if toolset. *Lecture Notes in Computer Science Formal Methods for the Design of Real-Time Systems*, pages 237–267, 2004. 31

[16] G. Madl and S. Abdelwahed. Model-based analysis of distributed real-time embedded system composition. *Proceedings of the 5th ACM international conference on Embedded software - EMSOFT 05*, 2005. 31

[17] A. David, L. Jacobsen, M. Jacobsen, K. Y. Jrgensen, M. H. Moller, and J. Srba. TAPAAL 2.0: Integrated Development Environment for Timed-Arc Petri Nets. *Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science*, pages 492–497, 2012. 31

Appendix A

LED model

This is the mCRL2 LED model generated by the model transformation system from an ASD specification. This model is manually modified to function properly. The LED model contains a *SwitchOn* button and *SwitchOff* button. In the initial state, the LED is off, and after the *SwitchOn* button is pressed, the Timer will be triggered. When the time is out, the LED light will turn on. Whenever the *SwitchOff* button is pressed, the timer will be cancelled, and the LED will switch off.

```
sort Enumeration = struct NoReplyValue |
                   asomeITimer'ITimer'VoidReply |
                   led'ILED'VoidReply;
sort LockingState = struct NONE | LOCK''led'sbs_led |
                    LOCK''asomeITimer'asomeBuiltinITimerSync'ITimerProtocol;
sort Notification = struct asomeITimer'ITimerCB'Timeout;
act asomeITimer'ITimer'CancelTimer;
act asomeITimer'ITimer'CreateTimer;
act asomeITimer'ITimer'CreateTimerEx;
act asomeITimer'ITimer'CreateTimerMSec;
act inevitableInternalTrigger;
act initialize;
act invalidate;
act invoke''asomeITimer'ITimer'CancelTimer;
act invoke''asomeITimer'ITimer'CreateTimer;
act invoke''asomeITimer'ITimer'CreateTimerEx;
act invoke''asomeITimer'ITimer'CreateTimerMSec;
act invoke''led'ILED'SwitchOff;
act invoke''led'ILED'SwitchOn;
act invoked''asomeITimer'ITimer'CancelTimer;
act invoked''asomeITimer'ITimer'CreateTimer;
act invoked''asomeITimer'ITimer'CreateTimerEx;
act invoked''asomeITimer'ITimer'CreateTimerMSec;
act invoked''led'ILED'SwitchOff;
act invoked''led'ILED'SwitchOn;
act led'ILED'SwitchOff;
act led'ILED'SwitchOn;
act lockQ: LockingState;
act lockQ_r: LockingState;
act lockQ_s: LockingState;
act optionalInternalTrigger;
act outwardNotification: Notification;
act outwardReply: Enumeration;
act pushNotification: Notification;
act qEmpty;
act qEmpty_r;
act qEmpty_s;
act qNonEmpty;
act qNonEmpty_r;
act qNonEmpty_s;
act queueSizeViolated;
```

```
act raiseNotification: Notification;
act readNotification: Notification;
act readReply: Enumeration;
act receiveNotification: Notification;
act sendNotification: Notification;
act sendReply: Enumeration;
act terminate;
act triggerNotification: Notification;
act unlockQ;
act unlockQ_r;
act unlockQ_s;
act valuedTrigger;
act writeReply: Enumeration;


proc Queue(q: List(Notification), locked: LockingState, t:Nat) =
      sum n: Notification . (sum x:Nat.(receiveNotification(n))@x .
      (Queue((q) <| (n), locked,x))) +
      (((#(q)) > (0)) &&
       (((locked) == (NONE)) || ((locked) == (LOCK''led'sbs_led)))) ->
        ((sendNotification(head(q)))@(t+1) .
         (Queue(tail(q), LOCK''led'sbs_led,t+1))) +
      sum s: LockingState .
      ((((#(q)) == (0)) && (((locked) == (NONE)) || ((locked) == (s)))) ->
       ((lockQ_r(s)) . (Queue(q, s,t)))) +
      ((locked) != (NONE)) -> (sum x:Nat.(unlockQ_r)@x . (Queue(q, NONE,x))) +
      ((#(q)) == (0)) -> ((qEmpty_r) . (Queue(q, locked,t))) +
      ((#(q)) > (0)) -> ((qNonEmpty_r) . (Queue(q, locked,t))) +
      ((#(q)) > (7)) -> ((queueSizeViolated) . (delta));

proc Terminate = (terminate) . (delta);

proc asomeITimer'asomeBuiltinITimerSync'ITimerProtocol'Active (t:Nat) =
    (true) -> ((invoked''asomeITimer'ITimer'CancelTimer) .
                writeReply(asomeITimer'ITimer'VoidReply) .
                (asomeITimer'asomeBuiltinITimerSync'ITimerProtocol'Inactive)) +
    (true) -> (((inevitableInternalTrigger) |
                (lockQ_s(LOCK''asomeITimer'asomeBuiltinITimerSync'ITimerProtocol)))
                @(t+40) .
                ((pushNotification(asomeITimer'ITimerCB'Timeout)@(t+41)) .
                ((unlockQ_s)@(t+42) .
                (asomeITimer'asomeBuiltinITimerSync'ITimerProtocol'Inactive))));

proc asomeITimer'asomeBuiltinITimerSync'ITimerProtocol'Inactive =
      (true) -> sum x:Nat.((invoked''asomeITimer'ITimer'CreateTimer)@x .
                writeReply(asomeITimer'ITimer'VoidReply)@(x+1) .
                (asomeITimer'asomeBuiltinITimerSync'ITimerProtocol'Active(x+2))) +
      (true) -> ((invoked''asomeITimer'ITimer'CancelTimer) .
                  writeReply(asomeITimer'ITimer'VoidReply) .
                  (asomeITimer'asomeBuiltinITimerSync'ITimerProtocol'Inactive)) +
      (true) -> (delta);

proc led'sbs_led'Off(replyValue: Enumeration) =
      (true) -> sum x:Nat.(1<=x&&x<=5)->
                 (((led'ILED'SwitchOn) | (lockQ_s(LOCK''led'sbs_led)))@x .
                 (((invoke''asomeITimer'ITimer'CreateTimer)@(x+1) .
                 (readReply(asomeITimer'ITimer'VoidReply))@(x+2)) .
                 (((qEmpty_s)@(x+3) .
                 ((outwardReply(led'ILED'VoidReply)@(x+4)) .
                 ((unlockQ_s)@(x+5) .
                 (led'sbs_led'SwitchingOn(NoReplyValue,x+5))))) +
                 ((qNonEmpty_s) .
                 (led'sbs_led'SwitchingOn(led'ILED'VoidReply,x+5))))))
    + (true) -> (delta)
    + (true) -> ((readNotification(asomeITimer'ITimerCB'Timeout)) . (Terminate));


```

```
proc led'sbs_led'On(replyValue: Enumeration) =
      (true) -> (delta)
    + (true) -> ((readNotification(asomeITimer'ITimerCB'Timeout)) . (Terminate));

proc led'sbs_led'SwitchingOn(replyValue: Enumeration,t:Nat) =
      (true) -> (delta)
    + (true) -> (((led'ILED'SwitchOff) | (lockQ_s(LOCK''led'sbs_led)))@(t+30) .
                  (((invoke''asomeITimer'ITimer'CancelTimer)@(t+31) .
                  (readReply(asomeITimer'ITimer'VoidReply))@(t+32)) .
                  (((qEmpty_s)@(t+33) .
                  ((outwardReply(led'ILED'VoidReply)@(t+34)) .
                  ((unlockQ_s)@(t+35)).(delta@(t+36)))) +
                  ((qNonEmpty_s)))))
    + (true) -> sum x:Nat.((readNotification(asomeITimer'ITimerCB'Timeout)@x) .
                           (((invoke''asomeITimer'ITimer'CancelTimer)@(x+1) .
                           (readReply(asomeITimer'ITimer'VoidReply)@(x+2))) .
                           (((qEmpty_s)@(x+3) .
                            ((((replyValue) != (NoReplyValue)) ->
                            ((outwardReply(replyValue)@(x+4)) .
                             (unlockQ_s)@(x+5)) <> (unlockQ_s)@(x+4)) .
                             (led'sbs_led'On(NoReplyValue)))) +
                            ((qNonEmpty_s) .
                            (led'sbs_led'On(replyValue))))));

init allow({initialize, invalidate, terminate, valuedTrigger,
            optionalInternalTrigger|lockQ, inevitableInternalTrigger|lockQ,
            led'ILED'SwitchOff|lockQ, led'ILED'SwitchOn|lockQ, outwardReply,
            outwardNotification, asomeITimer'ITimer'CancelTimer,
            asomeITimer'ITimer'CreateTimer,
            asomeITimer'ITimer'CreateTimerEx,
            asomeITimer'ITimer'CreateTimerMSec, sendReply,
            triggerNotification, raiseNotification, unlockQ,
            queueSizeViolated, qEmpty, qNonEmpty},
    comm({readReply|writeReply -> sendReply, lockQ_s|lockQ_r -> lockQ,
          unlockQ_s|unlockQ_r -> unlockQ, qEmpty_s|qEmpty_r -> qEmpty,
          qNonEmpty_s|qNonEmpty_r -> qNonEmpty,
          sendNotification|readNotification -> triggerNotification,
          pushNotification|receiveNotification -> raiseNotification,
          invoke''asomeITimer'ITimer'CancelTimer|
          invoked''asomeITimer'ITimer'CancelTimer ->
          asomeITimer'ITimer'CancelTimer,
          invoke''asomeITimer'ITimer'CreateTimer|
          invoked''asomeITimer'ITimer'CreateTimer ->
          asomeITimer'ITimer'CreateTimer,
          invoke''asomeITimer'ITimer'CreateTimerEx|
          invoked''asomeITimer'ITimer'CreateTimerEx ->
          asomeITimer'ITimer'CreateTimerEx,
          invoke''asomeITimer'ITimer'CreateTimerMSec|
          invoked''asomeITimer'ITimer'CreateTimerMSec ->
          asomeITimer'ITimer'CreateTimerMSec,
          invoke''led'ILED'SwitchOff|invoked''led'ILED'SwitchOff ->
          led'ILED'SwitchOff,
          invoke''led'ILED'SwitchOn|invoked''led'ILED'SwitchOn ->
          led'ILED'SwitchOn},
      ((Queue([], NONE, 0)) || (led'sbs_led'Off(NoReplyValue))) ||
       (asomeITimer'asomeBuiltinITimerSync'ITimerProtocol'Inactive)));
```

The interface model and the implementation model of the LED ASD model are shown in Figure A.1 and Figure A.2:

	Interface	Event	Guard	Actions	State Variable Updates	Target State	Comments
1	**Off (initial state)**						
2		StateInvariant		-		-	
3	ILED	SwitchOn		ILED.VoidReply		On	*LED is turned on*
4	ILED	SwitchOff		Illegal		-	
5	**On**						
6		StateInvariant		-		-	
7	ILED	SwitchOn		Illegal		-	
8	ILED	SwitchOff		ILED.VoidReply		Off	*LED is turned off*

Figure A.1: The specification of an LED system

	Interface	Event	Guard	Actions	ıte Variable Updat	Target State	Comments
1	**Off (initial state)**						
2		StateInvariant		-		-	
3		DataInvariant		-		-	
4	ILED	SwitchOn		ILED.VoidReply; Timer:ITimer.CreateTimer(5)		SwitchingOn	*Create a 5 second timer*
5	ILED	SwitchOff		Illegal		-	
6	Timer:ITimerCB	Timeout		Illegal		-	
7	**SwitchingOn**						
8		StateInvariant		-		-	
9		DataInvariant		-		-	
10	ILED	SwitchOn		Illegal		-	
11	ILED	SwitchOff		ILED.VoidReply; Timer:ITimer.CancelTimer		Off	*LED will not be turned on and the timer is canceled*
12	Timer:ITimerCB	Timeout		Timer:ITimer.CancelTimer		On	*Timeout - turn the LED on*
13	**On**						
14		StateInvariant		-		-	
15		DataInvariant		-		-	
16	ILED	SwitchOn		Illegal		-	
17	ILED	SwitchOff		ILED.VoidReply		Off	*Turn off the LED*
18	Timer:ITimerCB	Timeout		Illegal		-	

Figure A.2: The implementation of an LED system

Appendix B

ASML Requirements

This is a confidential appendix that only included in the confidential report.

Appendix C

Specification of mCRL2 Models

Models in this appendix are used to demonstrate the applicability of mCRL2 timer process.

C.1 Signal indicator model

The signal indicator model corresponds to Section 4.4

```
act startTimer, startTimer_c, startTimerA, startTimerA_c,startTimerB,startTimerB_c,
    startTimerC,startTimerC_c;
act cancelTimer, cancelTimer_c, cancelTimerA, cancelTimerA_c,cancelTimerB,
    cancelTimerB_c,cancelTimerC,cancelTimerC_c;
act timeOut, timeOut_c;
act timeOutA,timeOutA_c;
act timeOutB,timeOutB_c;
act timeOutC,timeOutC_c;
act signalY:Bool;
act recoverDone;
act signalA:Bool;signalA_c:Bool;
act signalB:Bool;signalB_c:Bool;
act recoverADone,recoverBDone,reset;
act recoverADone_c,recoverBDone_c;


map countdown,countdownA,countdownB:Real;
eqn countdown = 120; countdownA = 100; countdownB = 80;

proc Timer = sum x:Real. startTimer_c@x.
             (timeOut_c@(x+countdown)+sum y:Real.
             (y<x+countdown) -> cancelTimer_c@y).Timer;

proc TimerA = sum x:Real. startTimerA_c@x.
              (timeOutA_c@(x+countdownA)+sum y:Real.
              (y<x+countdownA) -> cancelTimerA_c@y).TimerA;

proc TimerB = sum x:Real. startTimerB_c@x.
              (timeOutB_c@(x+countdownB)+sum y:Real.
              (y<x+countdownB) -> cancelTimerB_c@y).TimerB;

proc signalX(sA:Bool,sB:Bool,sT:Bool,t:Real)=
        (!sA&&!sT)-> (sum x:Real.signalA(false)|startTimerA_c|startTimer_c@(x).
                      signalX(true,sB,true,x)) +
        (!sB&&sA&&sT) -> (sum x:Real.signalB(false)|startTimerB_c@x.
                          signalX(sA,true,sT,x)) +
        (sT&&sB&&!sA) -> (sum x:Real.timeOut_c|cancelTimerB_c@(x).
                          signalY(false)@(x+1).reset@(x+2).
                          signalA(true)|signalB(true)@(x+3).
                          signalX(false,false,false,x+3))+
        (sT&&sB&&!sA) -> (sum x:Real.timeOut_c|timeOutB_c@(x).
```

```
                              signalY(false)@(x+1).reset@(x+2).
                              signalA(true)|signalB(true)@(x+3).
                              signalX(false,false,false,x+3))+
        (sB&&sT) ->  sum x:Real.timeOutB_c|signalB(true)@x.
                     signalX(sA,false,sT,x)+
        (sA&&sT) ->  sum x:Real.timeOutA_c|signalA(true)@x.
                     signalX(false,sB,sT,x)+
        (sA&&sB&&sT) -> sum x:Real.timeOutA_c|timeOutB_c@x.
                        signalA(true)|signalB(true).
                        signalX(false,false,true,x) +
        (!sA&&!sB&&sT) -> sum x:Real.recoverDone|cancelTimer_c@(x).
                              signalY(true)@(x+1).
                              signalX(false,false,false,x+1);

init allow({startTimer,startTimerA,startTimerB,timeOut,timeOutA,timeOutB,signalY,
            recoverDone,signalA,signalB,cancelTimerA,cancelTimerB,cancelTimer,
            reset,cancelTimerA|cancelTimerB,timeOut|cancelTimerB,
            timeOutB|cancelTimer,timeOutA|cancelTimer, startTimerA|startTimer,
            timeOutA|timeOutB, timeOutA|timeOutB|timeOut,timeOutA|timeOut,
            recoverDone|cancelTimer,signalB|startTimerB,
            signalB|startTimerB|timeOutA,signalA|startTimerA,
            signalA|startTimerA|startTimer,timeOut|timeOutB,signalA|signalB,
            timeOutA|signalA, timeOutB|signalB},
       comm({startTimer_c|startTimer_c->startTimer,
          timeOut_c|timeOut_c->timeOut,
          cancelTimer_c|cancelTimer_c->cancelTimer,
          startTimerA_c|startTimerA_c->startTimerA,
          timeOutA_c|timeOutA_c->timeOutA,
          cancelTimerA_c|cancelTimerA_c->cancelTimerA,
          startTimerB_c|startTimerB_c->startTimerB,
          timeOutB_c|timeOutB_c->timeOutB,
          cancelTimerB_c|cancelTimerB_c->cancelTimerB},
        signalX(false,false,false,0)||Timer||TimerA||TimerB
    ));
```

C.2 Multi-processors model

The multi-processors model corresponds to Section 4.5

```
sort s = struct A1|A2|D1|D2|C1|C2|notDone;
act startTimer1,startTimer2,startTimer3:Real;
act startTimer1_c,startTimer2_c,startTimer3_c:Real;

act cancelTimer1,cancelTimer2,cancelTimer3;
act cancelTimer1_c,cancelTimer2_c,cancelTimer3_c;

act timeOut1,timeOut2,timeOut3;
act timeOut1_c,timeOut2_c,timeOut3_c;

act commuT1_T2,commuT1_T3,commuT3_T5;

act programDone;

act startTask1:Real;startTask2:Real;startTask3:Real;
act Task1Done:Real;Task2Done:Real;Task3Done:Real;

map c1,c2,c3:Real;
map cT1_T2,cT1_T3:Real;
eqn c1 = 10;c2 = 20;c3 = 10;
eqn cT1_T2 = 5;cT1_T3 = 10;

proc Timer1 = sum x,c:Real. startTimer1_c(c)@x.(timeOut1_c@(x+c)).Timer1;
proc Timer2 = sum x,c:Real. startTimer2_c(c)@x.(timeOut2_c@(x+c)).Timer2;
proc Timer3 = sum x,c:Real. startTimer3_c(c)@x.(timeOut3_c@(x+c)).Timer3;

proc controller(t1:s,t2:s,t3:s,p1:Bool,p2:Bool,t:Real,st:Real) =
          (t1==notDone) -> startTask1(1)|startTimer1_c(c1)@(t+1).
                           controller(A1,t2,t3,true,p2,t+1,t+1) +
          (t1==A1)      -> sum x:Real.Task1Done(1)|timeOut1_c@x.
                           controller(D1,t2,t3,false,p2,x,st) +

          (t1==D1 && t2==notDone) ->
               (!p1 -> startTask2(1)|startTimer1_c(c2)@(t+1).
                       controller(t1,A1,t3,true,p2,t+1,st)+
                p2 -> commuT1_T2|startTimer3_c(cT1_T2)@(t+1).
                      controller(t1,A2,t3,p1,true,t+1,st)) +
          (t1==D1 && t2==A1)       ->
                sum x:Real.Task2Done(1)|timeOut1_c@x.
                controller(t1,D1,t3,false,p2,x,st) +
          (t1==D1 && t2==A2)       ->
                sum x:Real.timeOut3_c|startTask2(2)|startTimer2_c(c2)@(x).
                controller(t1,C2,t3,p1,true,x,st) +
          (t1==D1 && t2==C2)       ->
                sum x:Real.Task2Done(2)|timeOut2_c@x.
                controller(t1,D2,t3,p1,false,x,st) +

          (t1==D1 && t3==notDone) ->
                (!p1 -> startTask3(1)|startTimer1_c(c3)@(t+1).
                        controller(t1,t2,A1,true,p2,t+1,st)+
                 !p2 -> commuT1_T3|startTimer3_c(cT1_T3)@(t+1).
                        controller(t1,t2,A2,p1,true,t+1,st)) +
          (t1==D1 && t3==A1)       ->
                sum x:Real.Task3Done(1)|timeOut1_c@x.
                controller(t1,t2,D1,false,p2,x,st) +
          (t1==D1 && t3==A2)       ->
                sum x:Real.timeOut3_c|startTask3(2)|startTimer2_c(c3)@(x).
                controller(t1,t2,C2,p1,true,x,st) +
          (t1==D1 && t3==C2)       ->
                sum x:Real.Task3Done(2)|timeOut2_c@x.
                controller(t1,t2,D2,p1,false,x,st)+
          (t1==D1 && (t2==D1||t2==D2) && (t3==D1||t3==D2)) ->
                programDone@(t+1).
                controller(notDone,notDone,notDone,false,false,t+1,st);
```

```
65
66 init allow({startTimer1,startTimer2,startTimer3,
67              cancelTimer1,cancelTimer2,cancelTimer3,
68              timeOut1,timeOut2,timeOut3,
69              startTask1,startTask2,startTask3,
70              Task1Done,Task2Done,Task3Done,programDone,
71              startTask1|startTimer1, Task1Done|timeOut1,
72              startTask2|startTimer1, startTask2|startTimer2,
73              Task2Done|timeOut2,commuT1_T2|startTimer3,
74              timeOut3|startTask2|startTimer2,
75              startTask3|startTimer1, startTask3|startTimer2,
76              Task3Done|timeOut1,Task3Done|timeOut2,Task2Done|timeOut1,
77              commuT1_T3|startTimer3,timeOut3|startTask3|startTimer2},
78      comm({startTimer1_c|startTimer1_c->startTimer1,
79            timeOut1_c|timeOut1_c->timeOut1,
80            cancelTimer1_c|cancelTimer1_c->cancelTimer1,
81            startTimer2_c|startTimer2_c->startTimer2,
82            timeOut2_c|timeOut2_c->timeOut2,
83            cancelTimer2_c|cancelTimer2_c->cancelTimer2,
84            startTimer3_c|startTimer3_c->startTimer3,
85            timeOut3_c|timeOut3_c->timeOut3,
86            cancelTimer3_c|cancelTimer3_c->cancelTimer3},
87     controller(notDone,notDone,notDone,false,false,0,0)||Timer1||Timer2||Timer3
88      ));
```

C.3 Data flow model

The data flow model corresponds to Section 4.6

```
act startTimer1,startTimer2,startTimer3:Real;
act startTimer1_c,startTimer2_c,startTimer3_c:Real;

act cancelTimer1,cancelTimer2,cancelTimer3;
act cancelTimer1_c,cancelTimer2_c,cancelTimer3_c;

act timeOut1,timeOut2,timeOut3;
act timeOut1_c,timeOut2_c,timeOut3_c;

act startTask1,startTask2,startTask3;
act Task1Done,Task2Done,Task3Done;

map c1,c2,c3:Real;
eqn c1 = 15;c2 = 15;c3 = 10;

proc Timer1 = sum x,c:Real. (startTimer1_c(c)@x.timeOut1_c@(x+c1)).Timer1;
proc Timer2 = sum x,c:Real. (startTimer2_c(c)@x.timeOut2_c@(x+c2)).Timer2;
proc Timer3 = sum x,c:Real. (startTimer3_c(c)@x.timeOut3_c@(x+c3)).Timer3;

proc controller(e1,e2,e3,e4:Nat,t1:Bool,t2:Bool,t3:Bool,t:Real) =
        (e3 >= 1 && e4 >= 1 && !t1) ->
            startTask1|startTimer1_c(c1)@t.
            controller(e1,e2,pred(Nat2Pos(e3)),pred(Nat2Pos(e4)),true,t2,t3,t+1) +
        (e1 >= 1 && !t2)            ->
            startTask2|startTimer1_c(c2)@t.
            controller(pred(Nat2Pos(e1)),e2,e3,e4,t1,true,t3,t+1) +
        (e2 >= 1 && !t3)            ->
            startTask3|startTimer3_c(c3)@t.
            controller(e1,pred(Nat2Pos(e2)),e3,e4,t1,t2,true,t+1) +
        t1 -> sum x:Real.Task1Done|timeOut1_c@x.
            controller(succ(e1),e2,e3,e4,false,t2,t3,x+1) +
        t2 -> sum x:Real.Task2Done|timeOut1_c@x.
            controller(e1,succ(e2),e3,succ(e4),t1,false,t3,x+1) +
        t3 -> sum x:Real.Task3Done|timeOut3_c@x.
            controller(e1,e2,succ(e3),e4,t1,t2,false,x+1);

init allow({startTimer1,startTimer2,startTimer3,
            cancelTimer1,cancelTimer2,cancelTimer3,
            timeOut1,timeOut2,timeOut3,
            startTask1,startTask2,startTask3,
            Task1Done,Task2Done,Task3Done,
            startTask1|startTimer1,startTask2|startTimer2,
            startTask3|startTimer3,startTask2|startTimer1,
            startTask3|startTimer1,
            Task1Done|timeOut1,Task2Done|timeOut2,
            Task3Done|timeOut3,Task2Done|timeOut1,
            Task3Done|timeOut1,
            startTask3|startTimer2|startTask1|startTimer1},
     comm({startTimer1_c|startTimer1_c->startTimer1,
           timeOut1_c|timeOut1_c->timeOut1,
           cancelTimer1_c|cancelTimer1_c->cancelTimer1,
           startTimer2_c|startTimer2_c->startTimer2,
           timeOut2_c|timeOut2_c->timeOut2,
           cancelTimer2_c|cancelTimer2_c->cancelTimer2,
           startTimer3_c|startTimer3_c->startTimer3,
           timeOut3_c|timeOut3_c->timeOut3,
           cancelTimer3_c|cancelTimer3_c->cancelTimer3},
    controller(0,0,2,1,false,false,false,1)||Timer1||Timer3
     ));
```

C.4 Wafer stepper model

The wafer stepper model corresponds to Section 4.7

```
sort wafer = struct A|B|None|pA|pB;

act startTimer_A:Real # Real;
act startTimer_A_c :Real # Real;
act timeOut_A, timeOut_A_c, timeOut_B, timeOut_B_c;

act waferAReceived, waferADelivered, waferAReceived_c, waferADelivered_c;
act waferBReceived, waferBDelivered, waferBReceived_c, waferBDelivered_c;

act startTimer1,startTimer2:Real;
act startTimer1_c,startTimer2_c:Real;

act cancelTimer1,cancelTimer2;
act cancelTimer1_c,cancelTimer2_c;

act timeOut1,timeOut2;
act timeOut1_c,timeOut2_c;

act loadReady;
act loadReady_c;
act deliverReady;
act deliverReady_c;
act start;
act start_c;
act stop;
act stop_c;
act startProcessing;
act machining;
act processingDone;
act timerEnd;

map lowA,lowB,highA,highB,c1:Real;
eqn lowA = 190;
eqn lowB = 140;
eqn highA = 220;
eqn highB = 170;

proc UnfixedTimer_A = sum x,l,h:Real.startTimer_A_c(l,h)@x.
                      sum y:Real.(x+l<=y && y<=x+h)->timeOut_A_c@y.
                      UnfixedTimer_A;
proc Timer1 = sum x,c:Real. (startTimer1_c(c)@x.timeOut1_c@(x+c1)).Timer1;

proc Controller(wait:wafer,p:wafer,tA:Bool,tB:Bool,t:Real) =
    (wait==None && p==None) -> (waferAReceived@(t+1).
                                Controller(A,None,tA,tB,t+1) +
                                waferBReceived@(t+1).
                                Controller(B,None,tA,tB,t+1)) +
    (wait==A && p==None)    -> waferADelivered_c@(t+1).
                               Controller(None,A,tA,tB,t+1) +
    (wait==B && p==None)    -> waferBDelivered_c@(t+1).
                               Controller(None,B,tA,tB,t+1) +
    (wait==None && p==A)    -> startProcessing|startTimer_A_c(lowA,highA)@(t+1).
                               Controller(None,pA,tA,tB,t+1) +
    (wait==None && p==B)    -> startProcessing|startTimer_A_c(lowB,highB)@(t+1).
                               Controller(None,pB,tA,tB,t+1) +
    (wait==None && p==pA && tA==false)   ->
                     startTimer1_c(150). Controller(None,pA,true,tB,t+1) +
    (wait==None && p==pB && tB==false)   ->
                     startTimer1_c(200). Controller(None,pB,tA,true,t+1) +
    (wait==None && p==pA && tA==true)    ->
                     sum x:Real.timeOut1_c@x.
                     (waferAReceived@(x+1).
                      Controller(A,pA,false,tB,x+1) +
                     waferBReceived@(x+1).Controller(B,pA,false,tB,x+1)) +
```

```
65      (wait==None && p==pB && tB==true)     ->
66                          sum x:Real.timeOut1_c@x.
67                          (waferAReceived@(x+1).
68                          Controller(A,pA,tA,false,x+1) +
69                          waferBReceived@(x+1).
70                          Controller(B,pA,tA,false,x+1)) +
71      (p==pA || p==pB) ->
72                          sum x:Real.processingDone|timeOut_A_c@x.
73                          Controller(wait,None,tA,tB,x+1);
74
75
76  init allow({startTimer_A, timeOut_A,
77              waferAReceived, waferADelivered,
78              startProcessing|startTimer_A,
79              startTimer_A_c|processingDone, start,
80              stop,machining,processingDone, loadReady,
81              deliverReady, waferBReceived, waferBDelivered,
82              stop|startTimer_A, timerEnd, startProcessing,
83              processingDone|timeOut_A},
84       comm({startTimer_A_c|startTimer_A_c->startTimer_A,
85             timeOut_A_c|timeOut_A_c->timeOut_A,
86             startTimer1_c|startTimer1_c->startTimer1,
87             timeOut1_c|timeOut1_c->timeOut1,
88             waferADelivered_c|waferADelivered_c->waferADelivered,
89             loadReady_c|loadReady_c -> loadReady,
90             stop_c|stop_c|stop_c->stop,
91             deliverReady_c|deliverReady_c->deliverReady,
92             waferBDelivered_c|waferBDelivered_c->waferBDelivered},
93       Controller(None,None,false,false,0)||UnfixedTimer_A||Timer1
94       ));
```