



Evolution Management and Process for
Real-Time Embedded Software Systems

Examples for Component Certification

D.3.7

Edited by Markus Pister

12.01.2004

Version 1.0

Status: Final

Public Version



I T E A

INFORMATION TECHNOLOGY

FOR EUROPEAN ADVANCEMENT

This document is part of the work of the EUREKA Σ! 2023 – ITEA 00103 project EMPRESS.
Copyright © 2002-2003 EMPRESS consortium.

Authors/Partners:

Partner	Author	Email
Technische Universität München	Jewgenij Botaschanjan Andreas Fleischmann Markus Pister	botascha@in.tum.de fleischa@in.tum.de pister@in.tum.de
Jabil	Emile Felix Linde Loomans	Linde_Loomans@jabil.com

Document History:

Date	Version	Editor	Description
10.1.2004	1.0	Stefan Van Baelen	Public version based on internal version 1.1

Filename: D3.7_v1.0_Public_Version.doc.doc

Abstract

Certification has the goal to offer a general scheme that indicates the quality or compatibility of a component in respect to certain properties. Therefore two tasks have to be performed. First an indicator for the fulfilment of the property has to be determined and second a method to show that this indicator is fulfilled has to be presented.

This deliverable shows two approaches for component certification. The first approach is a model based certification for time. When a component has to be integrated into a bigger system its response time of the component has to be fast enough to allow an adequate response time of the whole system. Thus, rules for composition of components are defined and the certificate of the whole system can be derived from the certificates of the subcomponents. The presented approach is representative for a model based time-certification for components which are developed with the CASE tool AutoFOCUS and are integrated in layered architectures.

The second approach deals with component certification of reusability. In contrast to the first approach, which is more or less a technical manual for certification, this approach is focused on the integration of the activity of certification into an existing software engineering process. It describes a process model including roles and their tasks to certify a component for its use in several systems. The main points of the approach are the dedicated role of a test & integration architect, who is responsible for the certification and the application of templates as guidelines for the component developers. The example approach has been tested in the environment of a Diagnosis-Software-System within Jabil.

Keywords:

Certification, reusability, time behaviour, components

Table of Contents

Abstract	3
Table of Contents	4
1 Introduction.....	6
2 Methods for Component Certification on Efficiency.....	8
2.1 Overview	8
2.2 The Design Paradigm	9
2.2.1.1 Architectural View	10
2.2.1.2 Behavioural View	10
2.2.1.3 Data View	12
2.2.1.4 Time Semantics in AutoFOCUS	12
2.3 The Case Study – Airbag	12
2.3.1 The Design Model	13
2.3.2 Requirements for Time and Resource Behaviour.....	14
2.4 Managing Requirements.....	15
2.4.1 Classification of Time Requirements	15
2.4.2 Translating Time Requirements into Design Constraints	16
2.4.2.1 System Structure Level (SSDs)	16
2.4.2.2 Translation to STDs	17
2.4.2.2.1 Translating States	17
2.4.2.2.2 Translating Events	19
2.4.2.3 Translating Time	20
2.4.2.4 Results of the Translation	20
2.5 Component Composition	21
2.5.1 Certificate of a Component	21
2.5.2 Reference Architecture	21
2.5.3 Composition Process	22
2.6 Certification of Atomic Components	22
2.6.1 Assumptions and Preconditions	22
2.6.2 Starting Situation	23
2.6.3 Steps	23
2.7 Conclusion	24

3	Method for Component Certification on Reusability.....	25
3.1	Context.....	25
3.2	Established processes and techniques to be tailored or improved	26
3.3	Results: Tailored and Improved Processes	27
3.3.1	Requirements Engineering.....	27
3.3.2	Architecture	28
3.3.3	Module Development	28
3.3.4	Module Testing.....	28
3.3.5	Integration/Integration Testing.....	29
3.3.6	Release Management	29
3.3.7	DSW Way-Of-Work document	29
3.4	Results: Template	30
3.5	Future Steps.....	30
4	Conclusion	31
5	Appendix	32
	Literature.....	38

1 Introduction

The complexity of embedded systems increases constantly. In order to cope with this complexity, software projects do not produce their software from the scratch, but try to reuse existing software as far as possible. However, this fact forces the developers to rely on the quality of the software to be integrated in the project. In the context of EMPRESS we talk about component based systems and by this, the software to be integrated is a component. The quality of a component can mainly be described with non functional properties like reliability, efficiency or reusability (see [ISO9126]). To be able to decide, if a component fits in the system to be developed, the quality of the integrated components has to be known.

The idea of certification (as shown in Figure 1) is that the user of a component relies on the verification and testing results of a third person which is called certifier. By this, the user of a component can save the effort of checking the component and only has to read the certificate, in which the properties of a component are guaranteed. The certifier can be a physical person, but also the results of tools that check the software can be seen as certificates (see [SW01]). One essential point of certification is that the user has to trust in the certifier since if the certifier is not reliable, the components have to be checked by the user again and the certificate becomes useless.

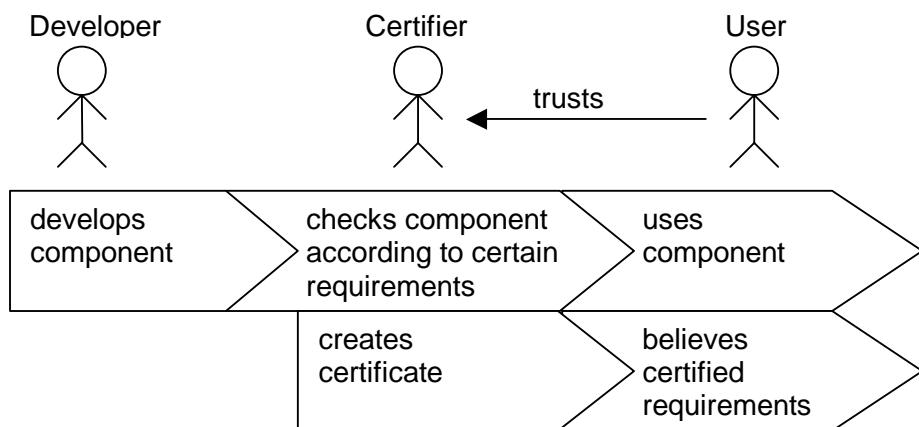


Figure 1: Certification in Software Projects

Certification has the goal to offer a general scheme used by all developers that indicates the quality or compatibility of a component in respect to certain properties. Therefore two tasks have to be performed. First an indicator for the fulfilment of the property has to be determined and second a method to show that this indicator is fulfilled has to be presented.

The indicator can be presented in many ways. In the following these ways are demonstrated with the example of reliability. One way is a “formal demonstration that a system or component complies with its specified requirements and is acceptable for operational use” [Emp03]. This means for example to check a direct metric like test coverage that allows a direct conclusion on the reliability. Another way is to check a certain development method with an emphasis on the reliability, like for example the V-Model. A detailed suggestion for reliability certification can be found in [IEC61508].

This deliverable shows two approaches for component certification. The first approach is a model based certification for time. When a component has to be integrated into a bigger

system its response time of the component has to be fast enough to allow an adequate response time of the whole system. The basic idea of the approach is similar to the idea of [GZ01]: Rules for composition of components are defined and the certificate of the whole system can be derived from the certificates of the subcomponents. The presented approach is representative for a model based time-certification for components which are developed with the CASE tool AutoFOCUS and are integrated in layered architectures.

The second approach deals with component certification for reusability. In contrast to the first approach, which is more or less a technical manual for certification, this approach is focused on the integration of the activity of certification into an existing software engineering process. It describes a process model including roles and their tasks to certify a component for its use in several systems. One main point of the approach is the dedicated role of a test & integration architect, who is responsible for the certification and the application of templates as guidelines for the component developers. The example approach has been tested in the environment of a Diagnosis-Software-System within Jabil.

The two presented approaches should not be seen as alternatives but as examples for complementary tasks in the field of certification. Some ideas of the approaches can be well combined (see Section 4).

2 Methods for Component Certification on Efficiency

2.1 Overview

In this chapter TU München presents an approach for the certification of components concerning their accordance to the specified time. The certification applies to the safety critical parts of the system (components) that have been identified during the risk analyses. The certification of a component is a formal demonstration that this component complies with its specified requirements and is acceptable for operational use [Emp03].

The presented approach suggests a method for the certification of systems with a component-based architecture. It covers the certification of the whole component architectures as well as of “atomic” components themselves. Figure 2 gives an overview over the activities and artefacts that are involved in the method.

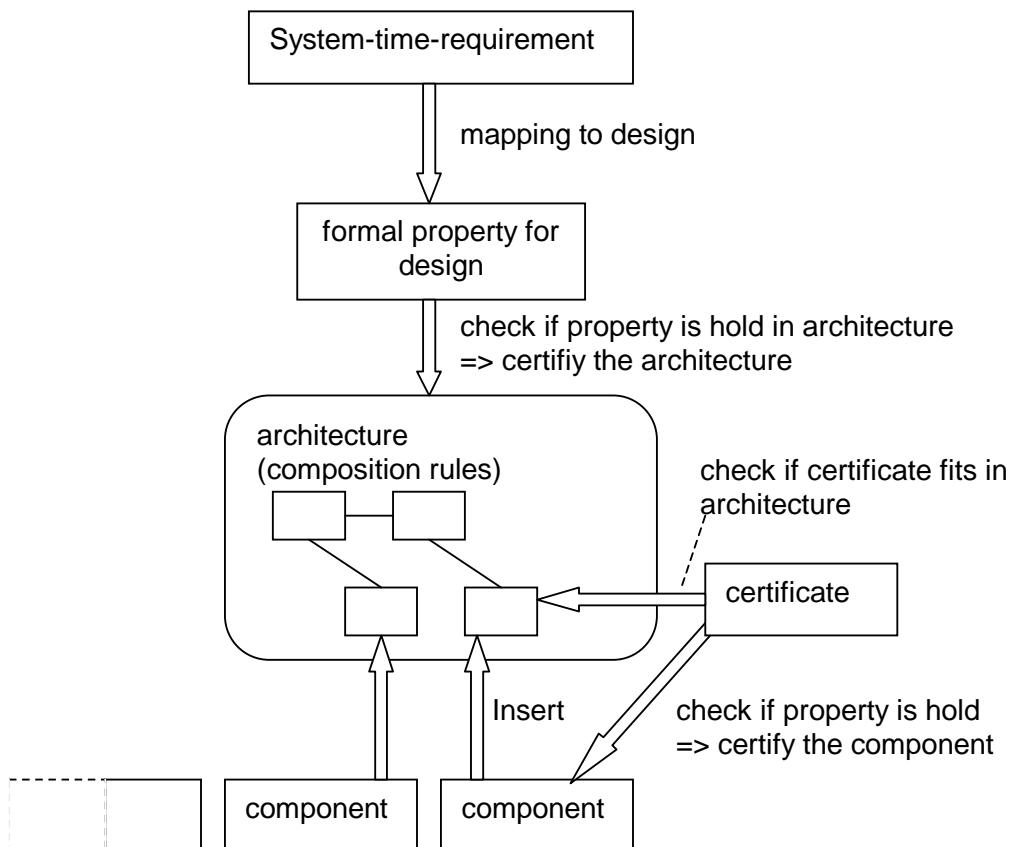


Figure 2: The Certification Approach

The components are designed using the AutoFOCUS paradigm (see [AutoFOCUS]). By this one task is to map the time-requirements to a formal description that can be verified in the system.

The verification of the architecture relies on the certificates of its constituent components. This permits the consideration of different design decisions or feasibility of substitution of components in a software system with reduced effort. The composition of the components to architectures obeys the certain rules, which limit possible type combinations and interaction

patterns between the components. This limitation eases the analysis of architectures and the exchange of components and allows the predictability of system's time behaviour and thus its certification.

The atomic components do not underlie these architectural restrictions. No assumptions about their internal design are made. The approach suggests certifying the components by checking, if the AutoFOSUC-design of a component satisfies the time requirements of the certificate. Since the description of components with AutoFOCUS is on a high abstraction level, the code generation and the deployment have to be considered as interfering factors.

The motivation for this approach is that there already exist tools and methods to check the consistency of an AutoFOCUS description and its implementation, for example by automatic test case generation, execution and coverage measurement as well as verified code generation algorithms. However, the compliance between a specification and a document on the abstraction level of AutoFOCUS can be guaranteed only by adducing additional manual effort.

The emphasis of the presented approach is the certification of time behaviour. However similar considerations can be made for other aspects of software systems, like resource consumption.

The next section (section 2.2) explains the design model of AutoFOCUS. The succeeding section (2.3) introduces a case study, which is the basis for the demonstration of the presented approach. Section 2.4 shows which relevant time requirements exist and how they can be mapped to the design. The next two sections cover methods to certificate the time properties of component architectures (section 2.5) and of the atomic components of the architecture (section 2.6). The conclusion at the end of this chapter summarizes the results and gives an outlook to future research work and applications.

2.2 The Design Paradigm

AutoFOCUS [AutoFOCUS] is a tool for graphically specifying embedded systems on the grounds of a simple, formally defined design paradigm. It supports different views on the model: structure, behaviour, interaction and data type view. Each view concentrates on a certain part of the specified model:

- The **architectural/structural view** shows, how the system is decomposed into components and by what ports and channels they are interconnected.
- The **behavioural view** describes the behaviour of each component by a finite state machine.
- The **data type view** defines the data types and functions to be used.
- The **interaction view** shows an exemplary set of interactions between components.

In the next, the views are described in detail. The certification approach is based on the structural, behavioural and data type views on the system. The interaction view is a basis for the test automation, which is used for the certification of the implementations. However, it is not directly used by the approach presented here. For details see [AutoFOCUS].

The AutoFOCUS tool and its design paradigm were chosen to demonstrate the abilities of the certification approach, because of the simple, precise semantics the tool is based on. However the approach can be adapted to other (graphical) description dialects (e.g. UML-RT) and tools, if they offer similar views on the modelled systems.

2.2.1.1 Architectural View

In AutoFOCUS a distributed system is modelled as a network of components, which communicate via channels. The architecture of a system is described with System Structure Diagrams (SSD). Figure 3 shows a typical SSD, which models a simple airbag controller. The elements of these diagrams are:

- **Components**, depicted as named rectangles. Components can be hierarchically decomposed. Every SSD shows an internal structure of a component, which consists of subcomponents and their interconnections. The leaves in this component hierarchy are called "atomic" components. Since such hierarchies can be flattened automatically, the presented approach concentrates on the atomic components, thus hierarchical SSDs won't be addressed in the remainder of this chapter.
 - **Ports** are special entry and exit points in the SSDs. They are visualized as solid circles (output port) or empty circles (input port) in the diagrams. Ports are typed and can be labelled with names. For example, "on-off-indicator:boolean" is a port called "on-off-indicator" of type "boolean". All ports attached to a component define the component's interface. Ports not related to a component in the diagram are meant to define the "outside world" and thus the component's interface to its environment.
 - **Channels** are depicted as directed named lines. Channels are typed and have a direction. They are connected to components at ports. A channel must connect exactly one input port with exactly one output port. The direction of a channel is always from its output to its input port. Note that unlike in UML-class-diagrams the connections between components transfer signals and are not used for referencing each other. Channels are named and typed using the same syntax as for ports. The type of a channel must equal the types of its ports.

In Figure 3 the component "Airbag Control" is connected to another component "Ignition" by two channels. The component "Airbag Control" is also connected with the environment's port via the channel "blast". This port represents the interface of the actuator of the modelled airbag, responsible for the blasting.

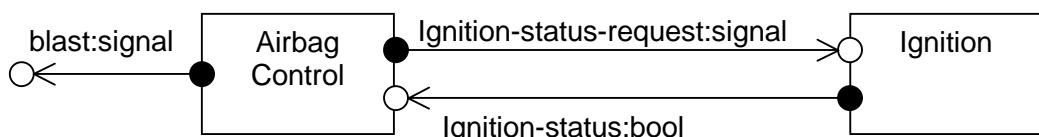


Figure 3: Example for a simple SSD

2.2.1.2 Behavioural View

The behaviour of an atomic AutoFOCUS component is described by an extended Mealy automaton. Its graphical representation is captured in State Transition Diagrams (STD). Figure 4 contains an example of a typical STD and SSD of an atomic component. STDs consist of the following elements:

- **Control states** are depicted by named ovals. The initial state is marked with a black dot.
 - The values of **local variables** build the data state of the component's automaton. Hence the internal state of a component consists of a control state and a data state. Local variables are declared using the QuestF language (see next section). The

syntax is (in EBNF [ISO14977]):

local_variable_definition ::= type : name = init_value ;

- **Transitions** are depicted as annotated arrows between states. If a transition starts and ends at the same state, it is called "self-loop". The label is annotated by a precondition, input patterns, output patterns and actions separated by colons. In addition, symbolic names can be used. The syntax of a transition annotation is:

transition_annotation ::= [symbolic_name] transition

transition ::= [precondition] : [input_patterns] : [output_patterns] : [actions]

The semantic of a transition is: if the input patterns do match and the precondition is true, then the output patterns evaluation and actions can be performed. So the execution of one transition can be subdivided into examination (input pattern + precondition) and firing (output pattern + actions). The examination condition can be empty, i.e. the transition can always be executed. The semantic and syntax of every part-construct of a transition is given below.

- **input_patterns** ::= input_port_id ? qterm [; input_patterns]

The corresponding transition is only executable if the value of the input port matches the pattern. For example "sensor-hw?ok" is true if and only if there is a value "ok" on the port "sensor-hw". If the pattern contains variables, they will be bound to their actual values and can be used later in expressions for precondition, output patterns and actions. Consider, for example, the pattern "p?Msg(x)". It expects a value of type "Msg" on port "p". This value will be written to a temporal variable "x", which can be used in other parts of transition. After the execution of the transition, its temporal variables are removed.

- **precondition** ::= boolean_qterm

A precondition is a Boolean expression containing local variables and temporal variables, bound in the input patterns. For the transition to be executable, the precondition must evaluate to "true".

- **output_patterns** ::= output_port_id ! qterm [; output_patterns]

Output patterns determine what values will be written to the specified output ports, when the transition is executed. For example, the execution of "alarm!false" will result in sending the value "false" through the port "alarm".

- **actions** ::= local_variable_id = qterm [; actions]

Actions assign values to local variables.

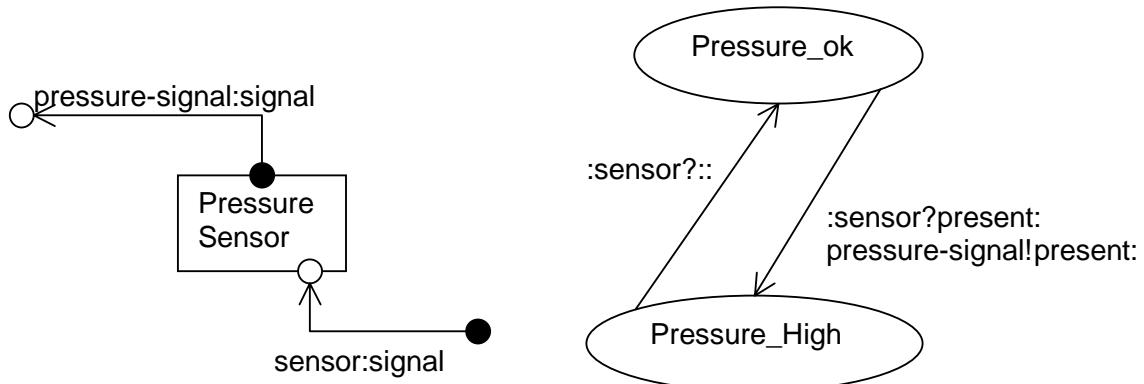


Figure 4: Example STD (right) and SSD (left) for a Component "Pressure Sensor"

An execution step of an AutoFOCUS component is equal to the firing of one transition of its automaton. The transition to be fired in the next state is chosen non-deterministically from the set of outgoing transitions of the current state with their examination evaluated to “true”. Note that each state has an idle transition that is activated each time when no other transition can be fired from this state. This transition needs not to be explicitly drawn in STDs.

2.2.1.3 Data View

The data view of a system is defined using QuestF, a functional language to describe data types and functions on them [QUG00].

QuestF has three predefined data types, namely Boolean, integer and float. In addition it allows defining complex hierarchical data types. The general form of a data type declaration is

```
dataType = Ci(sel11:Type11, ..., sel1n:Type1n) | ... |  
Cm(selm1:Typem1, ..., selmo: Typemo)
```

Thereby C_i is called "constructor". It can be used to construct a value of the corresponding type. A constructor can have an arbitrary number of arguments, including no arguments (constant). For each constructor a discriminator function is_C_i is generated. The sel_{ij} are called selectors. For a user-defined value, generated by a constructor term with arguments, each selector returns the appropriate argument. If no names are given for the selectors, AutoFOCUS automatically generates them by using the constructor names and a consecutive number. For example, an AutoFOCUS channel can be defined as a data type in the following way:

```
data Channel(alpha) = NoVal | Msg(val:alpha);
```

This definition means that a channel of type Channel(alpha) can either be empty (denoted by “NoVal”) or can carry a message of type “alpha” (where “alpha” is a type parameter, and is replaced by the actual type of the channel, e.g. integer).

The grammar rules for building QuestF terms are (in EBNF [ISO14977]):

```
qterm ::= qbase [: type]  
qbase ::= [ ± | - ] id |  
        ( qterm [ infid qterm ] ) |  
        if qterm then qterm [ else qterm ] fi |  
        unop ( qterm )
```

Whereas “unop” stands for an unary operator and “infid” stands for an infix operator. A list of all defined operators and a description of functional data types is given in Appendix A.

2.2.1.4 Time Semantics in AutoFOCUS

In the AutoFOCUS design model time is represented by discrete steps (called “ticks”) generated by a global clock. In each tick in every component the input values are read from input ports, one transition is fired and its outputs are written to output ports. The produced outputs are sent through channels to their destination ports exactly when the next tick begins. The values of ports are valid only for the duration of one tick. So the tick semantic models a synchronous communication with buffer length one between AutoFOCUS components.

2.3 The Case Study – Airbag

The certification of time and resource behaviour of components will be demonstrated with the

case study of a simplified airbag controller of a co-driver's seat in a car.

2.3.1 The Design Model

The interacting components are the airbag control itself, the ignition and the child lock controllers (see overview in Figure 5, and SSD in Figure 6), which communicate with an environment, build by the airbag actuator, which blasts the airbag, as well as pressure, ignition and child lock sensors.

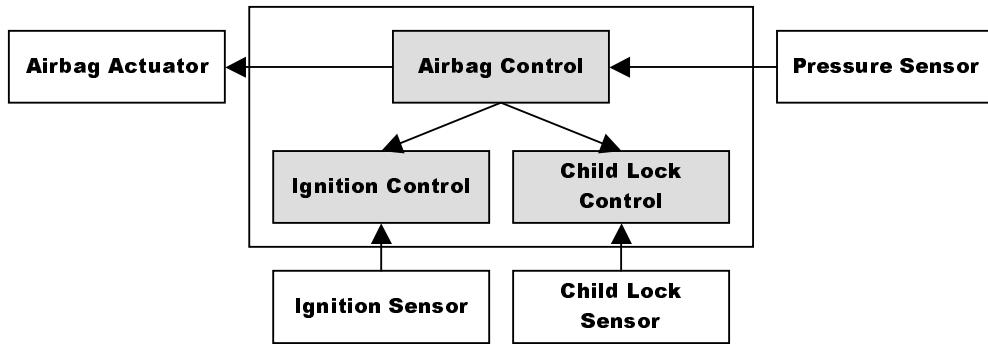


Figure 5: Overview of the Case Study

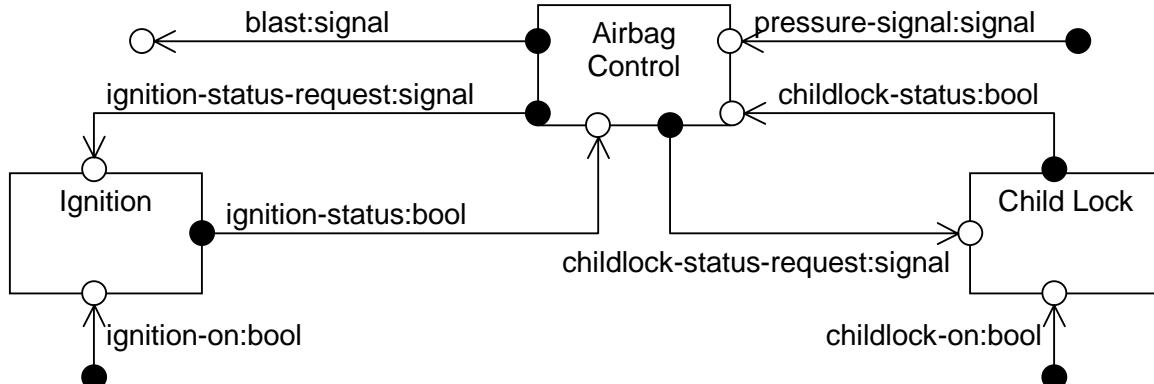


Figure 6: SSD of the Airbag Controller

The behaviour of this system is described as follows: If the pressure sensor signals a dangerous pressure on the car, the airbag control component will expand the co-driver's airbag if and only if the ignition is on and the child lock is deactivated. After the airbag has expanded, the control won't do anything any more. Figure 7 shows this behaviour in a STD.

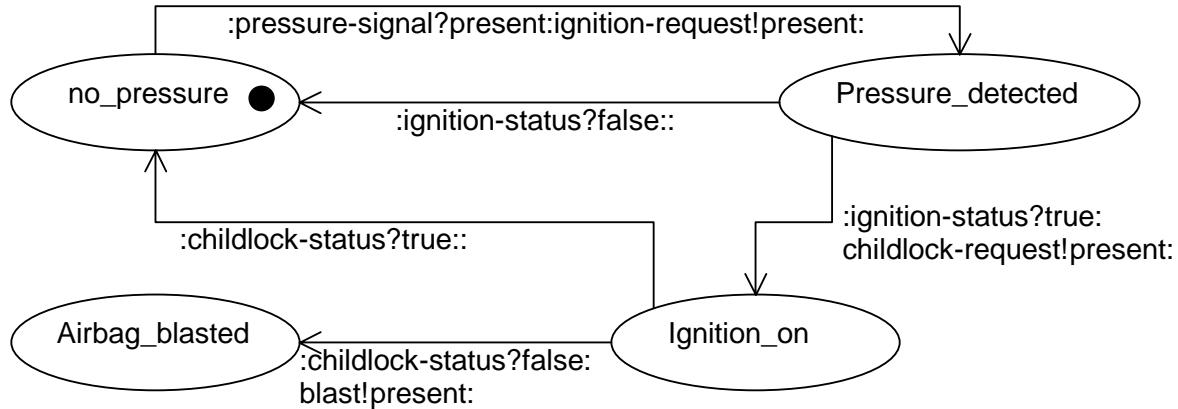
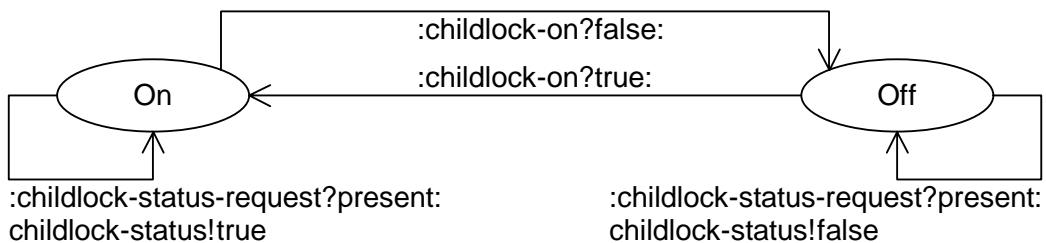
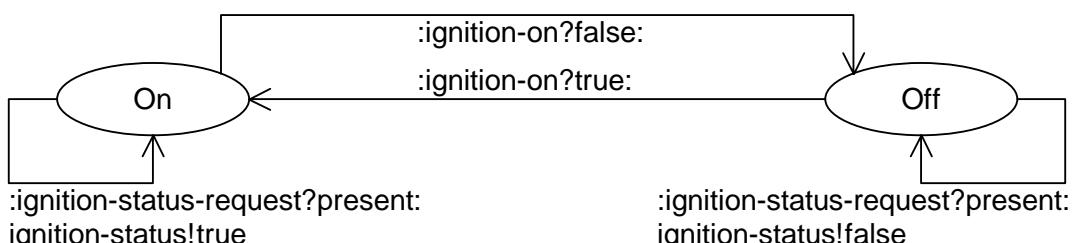
**Figure 7: STD of the Component Airbag Control.**

Figure 8 and Figure 9 show the state transition diagrams for the simplified ignition and child lock. They both have an identical behaviour: the ignition/child lock is activated as soon as a signal "true" arrives at the port "ignition-on" and "childlock-on". The component gets in the state "off", when a "false" arrives at these ports.

**Figure 8: STD of the Child Lock Component****Figure 9: STD of the Ignition Component**

2.3.2 Requirements for Time and Resource Behaviour

The specification of the airbag controller system consists of functional and non-functional requirements. Suppose the functional requirements were properly realized in the design of the system presented in the previous sections¹, the non-functional requirements concerning the

¹ The guarantee of this claim lies not in the scope of the presented approach.

time and resource behaviour of the airbag controller system have to be certified next.

Examples for time and resource requirements concerning the airbag system accompany the description of the certification approach in the remainder of this chapter. Here are some sample non-functional requirements:

- the time between ignition-on and initialization of the airbag control system must take at most 2 seconds
- after the pressure sensor signals a dangerous pressure, the airbag must expand in at most 1 second
- after the pressure sensor signals dangerous pressure, the airbag control system must check the ignition in at most 0.5 seconds.
- The airbag controller software is not allowed to allocate more than 400 Bytes of memory.

2.4 Managing Requirements

2.4.1 Classification of Time Requirements

Time requirements usually comprise two parts: the designation of a time interval, indicated by its start and end point, and propositions about the duration of this time interval. This information is often formulated implicitly and thus not always easy to determine. This section introduces a classification of the time requirements according to the different types of the time intervals and timing conditions.

There are two classes of timing conditions: absolute and relative. An orthogonal classification differentiates between conditions describing lower and upper bounds of time intervals. The classification of timing conditions is summarised in the following table:

	Upper bound	lower bound
absolute	in at most x seconds	in at least x seconds
relative	Not take x times longer as	at least x times longer as

Relative requirements can be translated in absolute requirements by measuring the length of the interval, with which it is compared. Due to this fact only requirements with absolute timing conditions are treated in the rest of this chapter.

The start and the end points of time intervals can be classified into three categories. This classification eases the later mapping into the design model of AutoFOCUS:

- **State-State:** The time interval is described by states as start and end points of the interval. For example: "the time between the ignition is on and initialization of the airbag control system must take at most 2 seconds."
- **Event-State/State-Event:** The time interval is described by an event as start point and a state as end point (or vice versa). For example: "after the pressure sensor signals dangerous pressure, the airbag control system must check the ignition in at most 0.5 seconds."
- **Event-Event:** Both the start point and the end point of the interval are events. For example: "after the pressure sensor signals dangerous pressure, the airbag must

expand in at most 1 second."

Templates for time requirements can ease the above classifications and thus facilitate the translation of time requirements into the terms of design model. Consider the following template. It requires that at first the start and end points are specified (that are states or events) followed by the information about the length of the time interval:

```
time-requirement ::= interval-def must be timing-cond
interval-def ::= the time between [ state | event ] and [ state | event ]
timing-cond ::= ( at least | at most ) abs-duration | rel-duration
abs-duration ::= x ( microseconds | milliseconds | seconds | minutes )
rel-duration ::= as long as interval-def | x times ( longer | shorter ) then interval-def
```

2.4.2 Translating Time Requirements into Design Constraints

The translation of time requirements into design constraints is done in three steps. Firstly, the requirements are translated into constraints which build on top of system structure diagrams (SSDs): those constraints refer to components, ports, channels, and abstract states. Secondly, the further translation to constraints building on top of state machines (STDs) is explained: those constraints refer to states, transitions, and ports. Additionally, the time mentioned in the requirement has to be translated into the tick semantic of the design model. The last subsection summarizes the results of the translation process.

2.4.2.1 System Structure Level (SSDs)

This subsection describes how time requirements are translated into the design level of system structure diagrams (SSDs). The four classes of time requirements as described in the previous section (state/state, event/state, state/event, and event/event) have to be mapped to components, port-valuations and abstract states. Therefore, requirements events and requirements states have to be translated.

A requirements event is translated into port valuations. For example, the time requirement "... the airbag must expand ..." can be translated into the port-assignment

$\text{valuation}_{\text{Airbag Control}}(\text{port}_{\text{blast}} == \text{present})$

A requirement state is translated into an abstract state which has to be defined. Consider the example: "the time between ignition-on and the initialisation of the airbag must take at most 1 second"; in this example, the requirements state "initialisation" can be mapped to "state_{initialized}" (see Figure 10). Abstract states concern the whole system and can be depicted as hexagons.

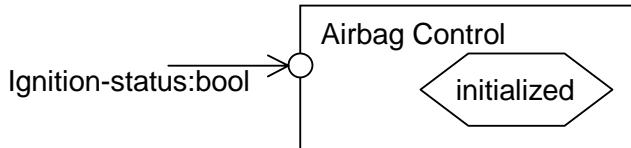


Figure 10: Translating a Requirements State to an Abstract State state_{initialized}

If the starting and the ending point of the interval described in the time requirement can be matched to port assignments, the corresponding time requirement can be written down as a predicate in the format

$\text{time}(\text{Start}; \text{End}; \text{duration})$

where Start describes the events and states that define the beginning of the interval and End describes its end. The syntax of a time constraint in the design level of SSDs has the following

format (in EBNF):

```

constraint ::= time( condition ; condition ; duration)
condition ::= (port-assignment | abstract-state) [, condition* ]
port-assignment ::= port portname (== | != | ≤ | ≥ ) value
abstract-state ::= state statename
duration ::= [ ≤ | ≥ | = ] value (microseconds | milliseconds | seconds | minutes )

```

For example, the time requirement "if the airbag gets a blast-signal, it must blast within 5 ms" can be translated to

```
time(portpres-in==present; portblasted==present; <=5ms)
```

It is also possible to mix port assignments and abstract states within the description of the start or end point of an interval. Time requirements can depend on preconditions, e.g. the time requirement "if the pressure sensors signal a car-crash, the airbag has to blast in 5 ms" might be accompanied by another requirement "the airbag must not blast if the ignition is off". In that case, abstract states can be used to map those preconditions as well:

```
time(portpressure=true, stateign_off; portblast=true; <=5ms)
```

Abstract states can also be used to express conditions that must hold for the time the predicate has to be valid. Another formal way to describe time requirements is a MSC (message sequence charts) that offers a great variety of possibilities to express such conditions. Usually these MSCs are translated afterwards to predicates like the ones presented in this section. Details on this topic can be found in [Kru00].

The result of this translation step is a list of abstract states and a set of predicates formulating time constraints between port assignments and abstract states.

2.4.2.2 Translation to STDs

2.4.2.2.1 Translating States

To denote a particular state of the system in the design level of state machines, a control state of each of its component has to be denoted. However, an abstract state description in the SSD level mostly describes only the states of a part of the system and makes no statements about the states of other components. Thus for all components, about which no statement is made in the abstract states, all control states have to be assumed. This means in particular, that one abstract state description is translated to a set of state combinations in the design.

For example, consider the three components "Airbag Control", "Ignition" and "Child Lock" in this chapter's case study. An abstract state description state_x meaning "if ignition on is on and child lock is off and pressure is ok" (see Figure 11) specifies exactly one state of the complete system. Therefore, there is exactly one design state corresponding to the abstract state description: { [ignition.on, childlock.off, airbagcontrol.pressureok] }. See the STD in Figure 12. However a description at this level of preciseness is not usual in requirements specifications, because they need knowledge about the whole system.

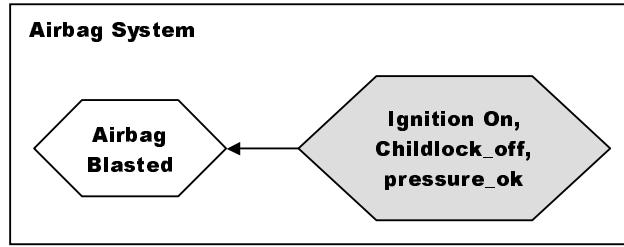


Figure 11: Precise State description at requirements level

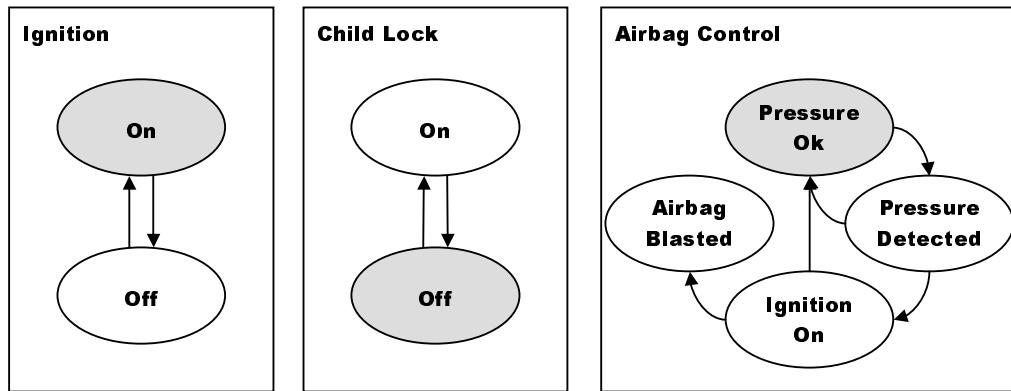


Figure 12: Mapping to one System State at Design Level

Most requirements refer to an abstract state of a system. If such an abstract state is given, only the precise given state information is used and for the other components, in which the states are not limited, every possible state combination is allowed. A state description like "The airbag is blasted" will be translated to four corresponding system states in the designed system:

```
{ [ignition.on, airbagcontrol.blasted, childlock.on];
[ignition.on, airbagcontrol.blasted, childlock.off];
[ignition.off, airbagcontrol.blasted, childlock.on];
[ignition.off, airbagcontrol.blasted, childlock.off] } See the STD in Figure 14.
```

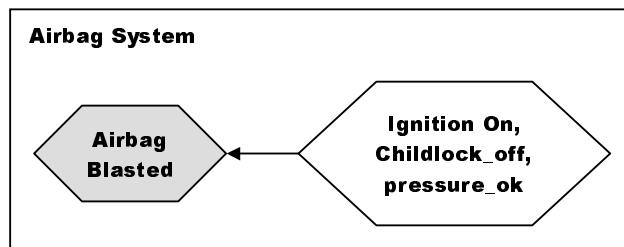
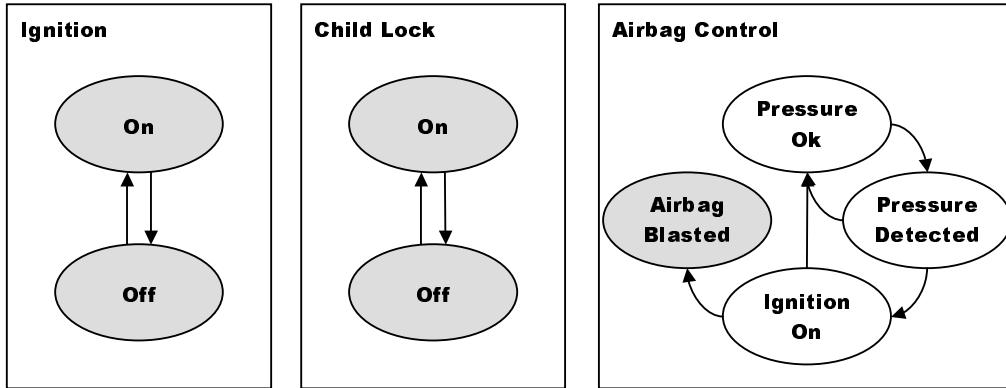


Figure 13: Abstract State Description at Requirements Level

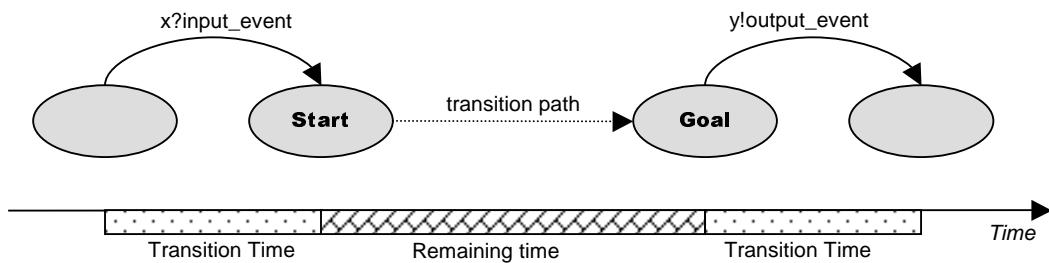
**Figure 14: Mapping to four System States**

In this example it is obvious, which states at design level correspond to an abstract state. However states at requirements level and abstract states are often described in an informal way. Here it is the task of the requirements engineer and the designer to determine the dependencies between the abstract states and the design states.

2.4.2.2.2 Translating Events

In the design level of the behavioural view, the notion of events is expressed by signals at ports. The signals can be sent by the component or obtained from its environment. Since events (e.g. signals) are produced or consumed by a component's automaton, they can be translated into a set of transitions of this automaton. A transition belongs to this set, if and only if the specified signal is written to the specified port in its output-patterns (in the case of incoming signals) or expected in its input-patterns (in the case of outgoing signals).

Finally, the source or destination states of these transitions for start or end points of the time interval respectively have to be determined (see Figure 15). They are the result sets of the translation, which can be handled by the model checking algorithm presented in section 2.6. The length of the time interval has to be reduced by the duration of the transitions, which were mapped to the event (since the length of the transition path is reduced by these transitions).

**Figure 15: Adapting the Time for Model Checking Algorithms**

For example, for the end point of a time interval described as the event "then the airbag has to blast" the corresponding signal is present on the port "blast". The affected transition is the transition between the states "Ignition On" and "Airbag Blasted". The resulting end state of the interval is the source state of this transition (see Figure 16).

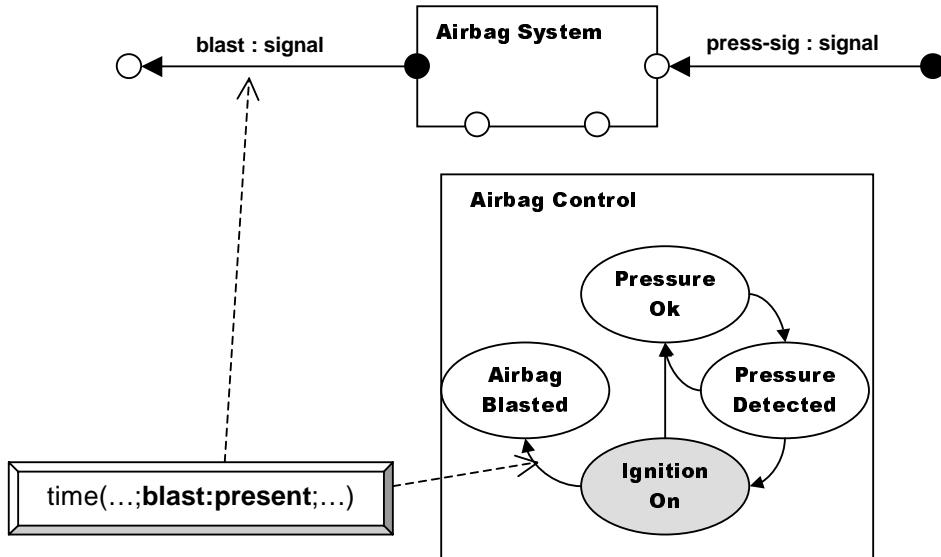


Figure 16: Mapping of Signals to Transitions and Starting States for Algorithms

An incoming event, which describes the starting point of a requirement's time interval, specifies an external event, which the component has to handle. If the set of states, which was obtained from an event, using the above procedure, is not equal to the set of all control states of the component, there are states in which the component cannot react on this event. It has to be analysed whether the occurrence of this event-state combination is possible and/or acceptable.

2.4.2.3 Translating Time

As explained in section 2.2.1.4, AutoFOCUS uses ticks to model time. However, the timing conditions of time requirements are usually expressed in conventional time units, like seconds or microseconds. The conversion between these two time systems can be performed using the following formula.

$$\text{number of ticks} = \left\lceil \frac{\text{time}}{\text{duration of one tick}} \right\rceil,$$

where the $\lceil . \rceil$ -function returns the least natural number, which is greater or equal than the function's argument. A method for the estimation of the duration of one tick is presented in Appendix B.

As mentioned in section 2.4.2.2.2, if the requirement uses an event to describe the start or end point of an interval, the translated interval duration has to be decreased by one tick.

2.4.2.4 Results of the Translation

The translation of a time requirement R results in

- a duration, expressed in ticks,
- an interval, defined by a set START(R) of tuples of starting control states and a set END(R) for tuples of end control states,
- if the states origin from a requirement event, the signal assignments to corresponding

ports are also reported.

The next section describes how to certify the time behaviour of a component based on this input.

2.5 Component Composition

This section discusses the derivation of certificates for the whole system from its (certified) subcomponents. The certification of “atomic” components is discussed in section 2.6 in detail.

2.5.1 Certificate of a Component

The goal of component certification is to guarantee its properties under particular environment conditions. Thus a certificate of a component consists of its properties and sets of feasible environments, for which these properties can be guaranteed. For the time behaviour, a certificate of a property is documented by

- The property itself (e.g. the non-functional requirement, constraining the execution time of some interaction pattern). See section 2.4 on classification and documentation of time requirements.
- The time interval $[a,b]$, which borders the execution times of the component’s implementation (of course this interval has to be conform with the requirement’s timing condition).
- The assumptions about the component’s environment, which are necessary, in order to guarantee the specified time behaviour. These assumptions can be also expressed in terms of time interval for the parts of the interaction pattern, described by the requirement, in which the environment of the component is involved (e.g. response times of the environment). This simple notion of environment assumptions presumes the independence of particular parts of an interaction pattern. This means, that the response times of calls to the environment can be distributed arbitrary. The only constraint is that the accumulated time consumption lies within the interval.

For a system, composed by a number of certified components, this guarantee can be derived from the certificates of its constituents. The components can be composed according to a specific composition environment (like in [GZ01]). In a composition environment the way of composition and interaction patterns are of interest (see next section). The resulting certificate consists again of guaranteed properties and guaranteeing environments for the system. This ensures the scalability of the presented approach.

2.5.2 Reference Architecture

In order to keep the modularity of the composed system, only a limited number of composition and interaction patterns are allowed by the presented approach. Therefore, a layered architecture is chosen as a reference. It organizes components of a system to several layers, numbered from 0 to n, and allows channels between a pair of components only if the layer number of the called component is smaller than the layer number its caller belongs to. Communication within the same layer is prohibited also. The system environment builds the top level layer (e.g. layer number 0). So in the remainder of this section the term “component” stands also for the system environment, which can easily be simulated as a layer 0 component.

2.5.3 Composition Process

Composing components to a compound component (system) or replacing one component by another arises two questions, which have to be answered in order to write out a system certificate:

- Are the assumptions on the component's environment met?
- Are the properties, demanded in the system specification fulfilled by the created component architecture?

The new environment sector of a component, which is interesting for the first question, consists of its communication partners from the lower layers. At the end of the call the results are transmitted back to the caller. The invocation can occur synchronous or asynchronous. However it is always synchronised by giving back the invocation results. For the caller component there exists an assumption of minimal/maximal response time of its environment. This assumption must subsume the sum of corresponding commitments of the called components. In other words, the interval in which the sum of response times of the called components lies has to be contained in the corresponding interval of the feasible environments of the caller component. The distribution of time consumption can be described for a pair of synchronous calls as the sum of their invocation time intervals and for asynchronous as the interval bounded by the minimal and maximal invocation times.

The second question from the above list is equal to the certification of the top-level environment components, using technique from the previous paragraph. This circumstance is explained by the characteristics of the chosen reference architecture: The duration of a particular interaction pattern of a component depends only on its communication partners from lower levels. As stated in the previous sections, the environment can be seen as one or several components. And the requirements on system time behaviour can be assigned to these components as the assumed time intervals.

2.6 Certification of Atomic Components

There already exist many verification methods for AutoFOCUS components [QUG00]. Certification algorithms should be clear to understand, because their purpose is to convince certifying institutes of the correctness of components. This section introduces one method based on [Wim00]. To enable a time certification that could be explained to others, the approach does not analyse the exact execution time but calculates the upper/lower bounds for the worst/best case. The verification of the property to be certified is done by a model checking algorithm.

The approach takes as input the time requirements that should be fulfilled. They can either be given as direct requirements from the specification or as consequence of the composition environment of the architecture.

The certification of a component is performed in two steps: The AutoFOCUS models are translated to a formal language for verification systems and the time properties are formulated as temporal logic formulas for the verification system. Finally the model checking algorithms check these demanded properties for the system.

2.6.1 Assumptions and Preconditions

The algorithm makes some assumptions about the model interpretation, the model design and the implementation of the model to deliver a correct result. These assumptions are limiting the

implementation and design variety but are still that small, that real world modelling practices are possible. In detail the assumptions are the following:

- **Ports are connected to only one channel.**

In AutoFOCUS, outgoing ports in STDs may be connected to several channels. The suggested algorithm merges different automata. Therefore it is easier to handle only ports, which are connected to only one channel. Diagrams that do not fulfil this condition can be transformed by adding new outgoing ports to components, which have the same behaviour as the original port. By performing this adaptation no information is lost.

- **All transitions need the same time to switch (global tick).**

This algorithm assumes that there exists a global timer that synchronizes the transitions. This assumption makes a simple merge of components possible, because of their synchronic execution.

2.6.2 Starting Situation

The input for the algorithm consists of two parts. One part is a set of assumptions $\{a, b, \dots\}$ about the calculation time and response time of the environment. In this approach the behaviour of the environment has to be described with automata which interact with the given component. This model of the environment forces the developer to think about things like user behaviour as uncertain factors for a certification. The automata for the environment are treated by the algorithm like the other components to be certified.

The second part of the input given to the algorithm is the translated requirement. This part consists of two sets defining the beginning and ending states for the requirement and of a time span, expressed in ticks. Additionally the valuation of the input ports can be given, if e.g. an event is the starting point of requirement's interval.

2.6.3 Steps

This section outlines the execution of the model checking algorithm.

The first step is to build one big component with one automaton that represents the behaviour of the system and the environment. This means a big component is built that simulates a closed world containing the behaviour of the component to be certified and of the behaviour described in the certificate for the environment.

In the second step, the merged automaton is converted into a so called Kripke structure [Cla01] (or it's more convenient representation by OBDDs²). In a simplified way, a Kripke structure describes all possible execution traces of an automaton. The translation from an AutoFOCUS automaton to an equivalent Kripke structure is described in [Wim00].

The last step is the application of a model checker. The Kripke structure, described in the previous section, serves as an input of a (symbolic) model checking algorithm. Another input is a time property, which has to be checked for the system being certified. In order to model check these properties, they have to be expressed by means of temporal logic. A special derivative of CTL³, specially suited for this purpose is real time CTL (RTCTL), presented in

² Ordered Binary Decision Diagrams

³ Computation Tree Logic

[Cla01].

Using RTCTL allows statements about the length of an execution path and thus about the number of ticks a system will need to reply on a request. The time between two ticks has to be offered by the developers. Appendix B gives a specific estimation for AutoFOCUS Systems.

2.7 Conclusion

This chapter presents an approach for the certification of time behaviour of components that are developed with AutoFOCUS: At first, it describes how the requirements are translated in terms of a AutoFOCUS model, and then it shows how they are checked in a composition environment.

We would like to thank Oscar Slotosch for his comments and suggestions.

3 Method for Component Certification on Reusability

For Jabil Hasselt, a first step towards and a prerequisite for component certification is the use of development processes and techniques which facilitate component reusability and the activities for ensuring this reusability.

In the context of EMPRESS, Jabil Circuit Hasselt has improved and tailored its development processes to support the development of reusable components as a first essential step towards component certification.

3.1 Context

Needs

One of the teams within the software development group of Jabil Hasselt is the Diagnostic Software (DSW) team. The task of the DSW team of Jabil Hasselt is to develop diagnostic software for consumer electronics products.

The purpose of diagnostic software (DSW) is performing diagnostic testing mainly on hardware and on PCB board interconnection of these products. For each product, three groups of diagnostic tests (diagnostic software) have to be developed:

1. tests to be performed during development and during production in the factory
2. tests to be performed in the field by service organisations
3. tests to be performed in the field by end users.

For each newly developed product at Jabil Hasselt, qualitative diagnostic software has to be developed.

The DSW projects have to cope with several problems:

Firstly, the development of DSW has to be done within the limited time frames of the development projects. Moreover, it is essential that the DSW is available as early as possible in the development of a new product. Thus timing constraints are especially for DSW very strong.

And besides this, in the domain of consumer electronic products, the complexity of the product functionalities is always growing.

To face these problems -- the growing complexity and the high time pressure -- it is essential for the diagnostic software team to decrease the development times.

On the other hand, DSW systems in general consist of many, relatively small but similar components.

In this context and as a possible answer to these problems, the team started to investigate possibilities for development and certification of reusable DSW components. How to ensure reusability of components is a major issue in the area of reusable component development. For Jabil Hasselt, the starting point of the investigation was that the possibility of component certification depends highly on the way the components are developed. Development processes and techniques supporting reusability, will also facilitate component certification. With this background, Jabil concentrated in this investigation on process support for reusable component development as an essential step towards component certification.

Starting Point

Jabil Hasselt has already built up experience in the area of the process development. We refer here especially to the work of Jabil (previously as Philips Hasselt) in the context of the ITEA @Terminals project [Ter02].

The Jabil activities for T3.7 build further upon the Jabil results of ITEA @Terminals: the processes and techniques developed in this project have been tailored and improved towards the needs of reusable component development and component certification.

3.2 Established processes and techniques to be tailored or improved

In a first attempt to develop reusable DSW components, the DSW team worked according to the established processes and techniques. After these first attempts to develop reusable components, it turned out that several processes had to be tailored or improved in order to support reusable component development and component certification.

As already mentioned, a DSW system usually consists of a large number of components. In general these components have rather limited development times and are similar in different DSW systems. On the other hand, the established processes and techniques support especially the development of new software systems, consisting of a limited number of components, developed from scratch.

It turned out that with the usual ways of work large development times are especially consumed by the module development and the module test activities. As in general the requirements for similar modules in new DSW systems do not change, reusability would decrease development time. But this reusability turned out to be poor with the existing processes. Since the processes were not set-up to include reusability, and due to the large number of components, developers did not concern about this issue. This resulted in a variety of modules engineered according to different developer's styles. Even system related software issues were included in these modules. Reuse of modules developed with this way-of-work in new systems was very difficult.

Another conclusion was that with the usual processes, too much major software errors are discovered at the end-phase of the DSW, during the release management phase. This was a serious risk for the project planning. Reducing the time for release management was an important aim.

Processes supporting reusable component development and component certification could save much time and had to be investigated.

After these first experiences, the following processes have been selected for improvement or tailoring:

1. Requirements engineering:

Reuse of requirements is not supported by the existing processes. It is essential that from the beginning of the project, in the requirements phase reusable requirements are identified.

2. Architecture: assigning customer requirements to an architecture and architectural modules and a software system connecting the modules; definition of the software environment and environment related issues (e.g. choice of operating system and software operating conditions) interaction with hardware platform defined by used components and hardware-software interface.

With the established processes too much architecture-related tasks are mixed up with the development and the integration of the separate modules. For reusable component

(module) development, it is essential that module development is independent from system creation and integration aspects.

3. **Module development:** the development - design and implementation - of a SW module.

With the established processes, too much time is spent on module development. Main reason is that the module developers' responsibilities are not clear and that too much time needs to be spent to system related problems

4. **Module testing:** test if the module functionality matches the customer requirements. This is done on module level only; the system environment does not yet exist. Here it is only possible to check if the information passed to the module (data input) is handled correctly by verifying the module response (data output). Usually this means interaction with other non-existing modules which need to be stubbed.

As holds for module development and for the same reasons, with the established processes, too much time is spent on module testing

5. **Integration and integration testing:** the system integrating all developed modules is created. Integration testing consists of tests to verify that data is passed properly between the system and the modules and between modules themselves.

With the established processes, the time for integration and integration testing was very limited and only a few errors could be discovered in this phase. Main reason was that too much time was spent to the module development and module test phases while the release date is always a fixed deadline.

6. **Release management:** this process usually consists of two test phases; the α-test and the β-test. The α-test verifies the customer requirements on "white-box" approach. The β-test is a "black-box" approach. While α-test is executed with in depth knowledge of the software system, focussing on software critical issues, β-testing aims at verification from a customer point of view. It is an in depth test to verify if the system will pass the customer's usage of the product. Errors found during these two test phases are handled by a software-fault-report database and a repair process. After the product has an acceptable status, it will be released.

The main problem with the established processes was that too much major errors are discovered too late, in the release phase. These errors should be discovered earlier in integration phase.

3.3 Results: Tailored and Improved Processes

3.3.1 Requirements Engineering

As mentioned before, similar software modules of different DSW systems usually have similar requirements. The major improvement here was the assignment of a dedicated (DSW-) software architect. This architect is responsible for gathering requirements for a DSW system and for the definition of the architecture. He/she decides what can be reused and what needs to be newly developed in the development of a new DSW system.

A DSW system consists of several DSW modules which can be divided in three categories:

1. the driver software, which usually is externally delivered and included in the system;
2. the system SW modules, containing the SW defining the system: i.e. user interface definition, hardware-software interface tables and system specific diagnostic mode tables,

3. the software modules performing the test requirements as defined by the customer. These modules should be independent from any software system and free from driver dependent references and hardware related references.

The goal is to develop reusable software for the 3rd category of modules so that only the system SW and in some cases the driver SW have to be re-developed. These modules should be developed and tested once at module level, and further proofed to be fault-free during integration and integration test phase.

This way of work, existing requirements can just be integrated in the new system, bypassing their module development and test effort.

Of course, when new (non-yet-existing) test requirements are asked then these modules need to be re-developed as well.

3.3.2 Architecture

The system architecture process was improved by adding a dedicated test & integration architect. Together with the software architect, the test and integration architect forms the architecture team.

As in requirements engineering is described, the architect is responsible for defining the system architecture. The test & integration architect closely monitors these activities and is responsible for the creation of the system by integration of DSW modules and drivers in this system. He is also responsible for the integration test and release management are responsibilities.

The goal is to have a dedicated person for the creation and release of a DSW system.

By assigning the responsibilities of system creation and integration to dedicated architects, the software engineers only have to concern with the development and test of software modules. This is an important step towards a module-based development where reusability will be the main focus.

3.3.3 Module Development

Module development was improved by:

- assigning a software engineer as responsible for the complete development of a module; he/she only needs to focus on the module, not on system aspects.
- the creation of a template supporting the engineer to work towards reusable and certified modules, (see section 3.4.)

3.3.4 Module Testing

Module testing was improved by making the software engineer responsible for implementing and testing the module test requirements as well. The template (see 3.4) guides also the module test activities.

After successful execution of these tests, the module is ready to be integrated in a DSW system and the module will be passed to the test & integration architect for integration in a system.

3.3.5 Integration/Integration Testing

The integration and integration testing is done by the test & integration architect.

- **Integration:** modules are integrated a DSW system. This system is created by the test & Integration architect. The system should be mature enough for the testing of modules in a system environment.
- **Integration test:** this test will be done when a module is integrated. This is a test to ensure that the module interacts correctly with other modules and the system on the DSW hardware platform. This way after every time a module(s) is integrated a repeated incremental test of the system is done to decrease the number of failures to be found in release management significantly. Integration tests are carried out by the development engineers and are monitored by the test & integration architect. After integration test the status of the integrated modules is reported to the system architect. If the module passed this phase, the system architect will certify this DSW module or component as reusable and fault free.

Integration/integration testing should result in a yes/no for the certification of a component - in this case a SW module.

With the improved way of work for the previous phases (described in sections 3.3.1 -3.3.4) module related errors should be discovered during these phases. During system integration and integration testing it should prove that a component works well for all relevant DSW systems. To support this last step towards component certification, two major improvements have been defined for the integration process:

- Systems are tested in an incremental way. Failures can be found very early.
- Modules are tested in several systems at the same time. This is possible for DSW because DSW systems are usually created concurrent for different projects. This is an important test for the effective reusability of the component and as such a step forward towards component certification.

An important effect of this way of work is also that it eliminates the need for an α -test at the end of the DSW project.

3.3.6 Release Management

As stated before the Integration/Integration testing has been extended to a major phase in the development of a DSW system(s). The consequence is that there is no need for an α -test. What remains is the official release of the DSW system and hand over to the Quality Department for extensive β -testing.

The expectation is that the number of failures found in β -test will decrease and the failures related to module errors will also decrease.

3.3.7 DSW Way-Of-Work document

The improvements listed in previous sections have been described in a “DSW Way of Work document” [Fel03]. This document describes in detail what each team member should do in the creation of a successful DSW system. The focus is on the module development, module test, integration and integration testing phases, as these phases are the key phases in successful development of reusable modules. The document especially highlights the differences with the established processes and techniques, it can be considered as a tailored

process description.

3.4 Results: Template

To support the software engineer in the creation of software modules a template has been created. This template guides the engineer through the process of module development, while aiming at reusable and certified modules for DSW. The module document is to be reviewed by the architecture team before allowing the module to be integrated in any system.

The template consists of the following chapters:

- Requirements: an overview of the customer requirements fulfilled by this module.
- Overview of Module
 - Requirement identification: each requirement gets a unique identifier. This supports the system architect to have an overview of all requirements in a module. This way it is made easier to implement new DSW systems to see what which requirements are available and what needs to be newly created.
 - Design of module: the architect of the module consisting of conceptual architecture, execution architecture, interconnection - and code architecture. Also complex algorithms are described here and they are proven by predicates.
 - Test specifications of module: the set of testcases defined to meet each requirement. Important is that the requirements are tested not in the actual DSW system but in a development system (on PC with stubs) to test independently of hardware platforms - for reusability.
 - Test results of module: a summary of test results, also remaining compilation warnings, static code analyser results and a summary of remaining recommendations and conclusions;
 - metrics: metrics on code like size, number of code lines.

3.5 Future Steps

Next step is that the proposed improvements will be tried out in multiple DSW projects first, followed by other embedded SW development projects. When the practical use turns out to be satisfactory, the improvements will gradually become part of the standard processes for component-based development in Jabil.

With the described work, Jabil Hasselt intends to gain essential knowledge on component reusability and component certification, and to move an important step forward in this direction.

4 Conclusion

This document presented two examples for different tasks in the field of certification. One example is taken from the certification of time and the other one considers the certification of reusability. The first approach, dealing with time suggests a model-based development with automatic code generation. The models can be verified against time constraints. The approach is mainly focused on the certification technique and not on task to be performed in respect to project management and the software engineering process.

The second approach mainly deals with the process and the roles to enable and use a certification of reusability. The application of templates as rules for component design and the control from the dedicated role "test & integration architect" improve the implementation of certification activities in a software engineering process.

The process and role concept presented in the second example can be also used to integrate the certification technique of the first example. The observation of the developed components as well as the control of the composition mechanisms can be performed by the architecture group in the process.

5 Appendix

Appendix A Autofocus Data Types

This appendix shows the operators for data types of as well as a declaration of functional data types for AutoFOCUS. This appendix is relevant for the estimation of a tick duration that is described in Appendix B.

Operator	Symbol
equality	=
inequality	!=
receive	?
send	!
implication	=>
biimplication	<=>
disjunction	
conjunction	&&
subtraction	-
addition	+
multiplication	*
division	/
greater	>
greater/equal	>=
less	<
less/equal	<=

Table 1: Binary Operators in QuestF

operator	symbol
Negation	not
minus sign	-
plus sign	+

Table 2: Unary Operators in QuestF

AutoFOCUS data type declarations can contain user-defined functions. A user-defined function can be specified using different alternatives and pattern matching. Constants and functions can be declared in the following way:

```
decl ::= const id = qterm ; | fun qterm fundecls* ;
fundecls ::= | qterm
```

The “qterm” has to have the form “f(qterm) = qterm”, where f is the defined function, and in the left term only constructors and variables are allowed:

```
fun emptyChannel(NoVal) = true
| emptyChannel(Val(x)) = false;
```

Appendix B Estimation of Tick Duration

This section deals with the estimation of time consumption for AutoFOCUS systems. The constraints of time requirements are expressed in continuous units like seconds, microseconds etc. Section 2.2.1.4 pointed out that the time semantic of AutoFOCUS is based on a discrete notion of time, using the concept of ticks. In section 2.4.2.3 timing conditions of requirements are translated into the conditions on amounts of ticks. For this purpose the duration of one tick has to be estimated. Section B.2 gives this claimed estimation. Formulas for the time behaviour of different parts of AutoFOCUS models, needed for the calculation of tick duration, are given in a top-down manner in succeeding sections. But first some preliminary work on prioritization of transitions is made in section B.1. This prioritization will aid the estimations from the later sections.

B.1 Prioritization of Transitions

Every state S has a set of outgoing transitions $\text{TRANS}_S = \{t_S^{(1)}, \dots, t_S^{(n)}\}$, with $n > 0$. If the automaton gets in the state S , it examines each of its transitions until it finds one of them, which can be executed. The next state of the automaton is the destination of the executed transition. The transitions of S are examined in a certain order, according to the transitions' priorities.

Every transition is mapped to the sequence of priorities $\text{PRIO}_S = (p_S^{(1)}, \dots, p_S^{(m)})$ using the function $\text{prio}_S : \text{TRANS}_S \rightarrow \text{PRIO}_S$. This function defines a partition of the transition set TRANS_S to m priority classes $(PC_S^{(1)}, \dots, PC_S^{(m)})$ with $PC_S^{(i)} = \{t_S | \text{prio}_S(t_S) = p_S^{(i)}\}$ for all $1 \leq i \leq m$. The transition $t_S^{(i)}$ will be examined before the transition $t_S^{(j)}$ if $\text{prio}_S(t_S^{(i)}) < \text{prio}_S(t_S^{(j)})$. In the case the priorities are equal, the selection occurs non-deterministic. The lowest priority $p_S^{(m)}$ is reserved for a so called idle transition t_S^{idle} , which is executed every time, when no other transition can be fired. As pointed out in section 0, this transition is a self-loop, e.g. its source and destination are equal and it doesn't cause any changes on the automaton or on the component's environment.

Using the prioritization, the estimation of execution time can be easily adapted for different deployment types. For example in the case of code generation for a real-time embedded system, the examination of transitions will probably occur in a sequential, deterministic way. This fact can be mirrored by defining one priority class for every state's transition and ordering them according to the examination sequence. Another extreme is the fully non-deterministic choice of a transition to be examined. In this case there will be two priority classes, one for the idle self-loop (low) and another one for other transitions (high).

B.2 Estimation of a Tick Duration

The length of one tick for a given AutoFOCUS-system is defined as the time-span, in which every system's component can accomplish (e.g. start and finish) its execution step and its produced outputs are transported through channels to their consumers. This tick semantic, introduced in section 2.2.1.4, has to be implemented correctly, when deploying a modelled system to a specific platform. This section describes methods for the estimation of the tick duration for a certain class of deployment architectures.

In this section it will be assumed, that the model is implemented in a conventional programming language (e.g. ANSI C) and the execution of one automaton occurs strictly sequential and deterministic. The execution of the individual components can be accomplished in parallel as well as in sequential manner, depending on the underlying

deployment architecture. However the assumption for this deliverable is that one component is completely executed by one processor.

For a given system SYS consisting of a set of components $COMP_{SYS} = \{C_1, \dots, C_n\}$ with the associated set of automata $A_{SYS} = \{C.A \mid C \in COMP_{SYS}\}$, the following deployment architecture class $ARCH$ is treated in this section:

- The system is deployed to a set of processing units $PU_{ARCH} = \{PU_1, \dots, PU_m\}$, these can be processors or network nodes. This is done by a function $deploy : COMP_{SYS} \rightarrow PU_{ARCH}$. This function is total and surjective, e.g. it deploys every component to exactly one processing unit, and one processing unit can get more than one component.
- Channels are deployed to communication paths and protocols, which are available by and reasonable for the chosen architecture. This can be a special networking protocol in the case of a network deployment, semaphores and shared memory structures in multi-processor systems, or global variables for sequential execution on one processor. There is also a certain time demand for the synchronization between processing units in order to properly implement the tick semantic. The time needed for communication and synchronization is represented by a factor T_{comm} , which has to be calculated according to the chosen communication medias and protocols.

The estimation of the duration of one tick for the described class of systems is $T_{tick} = \max_{PU \in PU_{ARCH}} (T_{PU}) + T_{comm}$, i.e. it is the maximum of one computation step in a computing unit plus the transport and synchronization time.

It is assumed, that every processing unit executes its components in a deterministic sequential order. So its tick duration can be estimated as $T_{PU} = \sum_{C \in deploy^{-1}(PU)} T_{tick}^C + T_{comm}$, where

$deploy^{-1} : PU_{ARCH} \rightarrow \wp^{COMP_{SYS}}$ is the reverse deployment function, which returns for every processing unit a set of components deployed to it. In words, the tick duration can be estimated as a sum of ticks' duration of every deployed component plus the synchronization and transport time.

The estimation of duration of one tick for one component T_{tick}^C presented here aims to certify a certain time behaviour of the component C , as described in section **Error! Reference source not found..** This fact motivates to find a worst-case for the time behaviour. A worst-case, however, depends on the type of the property, which has to be certified. For upper-bounds the highest tick duration has to be considered, for lower-bounds the lowest one.

- In the case of an upper-bound constraint, the tick duration of an automaton A (and its parent component C) can be estimated using the following formula: $T_{tick}^C = T_{A_C} = \max_{t \in TRANS_{A_C}} (T_t)$, e.g. the tick duration is equal to the slowest execution of automaton's transition.
- For lower-bound constraints, the tick duration can be estimated by: $T_{tick}^C = T_{A_C} = \min_{t \in TRANS_{A_C} - \tau^{idle}} (T_t)$, with $\tau^{idle} = \{t_S^{idle} \mid S \in STATE_{A_C}\}$, e.g. the tick duration is equal to the quickest execution of component's transition, which is not an idle self-

loop.

An approach for the estimation of a transition execution time T_t is discussed in the following section.

B.3 Duration of Transition Steps

In order to estimate the execution time of one transition step, the syntactic and semantic structure of transitions has to be examined. Every transition t_s in AutoFOCUS has the following form:

transition ::= [precondition] ; [input_patterns] ; [output_patterns] ; [actions]

The semantic of such a transition is: if the input patterns do match and the precondition is true, then the output patterns evaluation and actions can be performed. So the execution of one transition can be subdivided into examination (precondition + input pattern) and firing (output pattern + actions). The overall execution time of a transition t_s is $T_{t_s} = ET_{t_s} + FT_{t_s}$,

where $ET_{t_s} = T_{\text{precondition}}^{t_s} + T_{\text{input_pattern}}^{t_s}$ and $FT_{t_s} = T_{\text{output_pattern}}^{t_s} + T_{\text{actions}}^{t_s}$. This estimation of examination time is a simplification, because in a case of pattern mismatch in one of the input patterns, the precondition has not to be examined and, further on, the transition will not be fired, if the precondition is not true or input patterns mismatch. A more elaborate estimation of execution time for transitions will be discussed next.

For every transition step $S_i \xrightarrow{t_{S_i}} S_{i+1}$ (read: the step from state S_i to state S_{i+1} using the transition t_{S_i}) the execution time is defined as

$$T_{S_i \xrightarrow{t_{S_i}} S_{i+1}} = \left(\sum_{j \leq \text{prio}_{S_i}(t_{S_i})} ET_{PC_{S_i}^{(j)}} \right) + FT_{t_{S_i}} \quad (1),$$

with $ET_{PC_{S_i}^{(j)}} = \sum_{t \in PC_{S_i}^{(j)}} ET_t$, i.e. the execution time is the sum of all examination times of all

transitions with a priority greater or equal than the priority of t_{S_i} plus the firing time of t_{S_i} itself. This estimation of execution time is a worst case, because it assumes that all transitions with the same priority as t_{S_i} (e.g. all $\hat{t}_{S_i} \in PC_{S_i}^{(j)} \wedge \hat{t}_{S_i} \neq t_{S_i}$, with $\text{prio}_{S_i}(t_{S_i}) = j$) are examined afore.

In order to get a better estimation for the lower-bound constraints discussed in the previous section, the following formula can be used:

$$T_{S_i \xrightarrow{t_{S_i}} S_{i+1}} = \left(\sum_{j < \text{prio}_{S_i}(t_{S_i})} ET_{PC_{S_i}^{(j)}} \right) + T_{t_{S_i}} \quad (2),$$

e.g. it is now assumed, that t_{S_i} is examined as the first transition from its priority class.

However, this estimation doesn't change anything with the assumptions about the deployment architecture made in the last section. Section **Error! Reference source not found.** mentions, that if the automata are executed sequentially and deterministic, the priority classes will be of the size one. So the two estimations (1) and (2) become equal in this case. Never the less, for other architecture classes, this estimation for lower-bound constraints could be useful.

B.4 Evaluation Time of Transition Expressions

This section deals with the duration estimation for the evaluation of an expression which annotates one transition. This estimation focuses on each of the four parts of this annotation (precondition, input pattern, output pattern and actions). These four constituents of a transition were introduced in section 0. They are formulated as QuestF-terms (see section 2.2.1.3 for details). In the following, their syntax in EBNF and semantic will be presented and formulas for estimating their evaluation time will be discussed.

- **input_patterns ::=** input_port_id ? pattern [; input_patterns]

The evaluation time of a list of input pattern is $T_{input_pattern} = \sum (T_{binotype}^{receive} + T_{qterm})$, e.g. the sum of receives and pattern matching operations for every member of the list. [Wim00] describes a static method for conversion of pattern terms into logical formulas. Using this method, the estimation above can be generalized to

$$T_{input_pattern} = \sum (T_{binotype}^{receive} + T_{qterm}).$$

These estimations assume a strict implementation language, i.e. ignore the fact that the evaluation of input patterns can be aborted as the first pattern mismatch is found.

- **precondition ::=** boolean_qterm
- **output_patterns ::=** output_port_id ! qterm [; output_patterns]

The evaluation time of a list of output pattern is $T_{output_pattern} = \sum (T_{binotype}^{send} + T_{qterm})$, e.g. the sum of send operations and evaluations of pattern expressions for every member of the list.

- **actions ::=** local_variable_id ≡ qterm [; actions]

The evaluation time of a list of actions is $T_{actions} = \sum (T_{assign} + T_{qterm})$, e.g. the sum of assignments and evaluations of right-hand expressions for every member of the list.

As aforementioned, the execution time of one transition is at most the sum of the evaluation time of its parts. The estimation of evaluation time for arbitrary QuestF-expressions (T_{qterm}) is discussed in the next section.

B.5 Evaluation Time of QuestF-Terms

According to the syntax of QuestF presented in section 2.2.1.3, the evaluation time of a QuestF-term T_{qterm} can be estimated depending on its type:

- $T_{qterm}^{if} = T_{qterm}^{cond} + \max(T_{qterm}^{then_branch}, T_{qterm}^{else_branch})$ for the if-expression in the case of the upper-bound estimation and $T_{qterm}^{if} = T_{qterm}^{cond} + \min(T_{qterm}^{then_branch}, T_{qterm}^{else_branch})$ for the lower-bound.
- $T_{qterm}^{binop} = T_{qterm}^{op1} + T_{binotype} + T_{qterm}^{op2}$ for binary operations.
- $T_{qterm}^{unop} = T_{unotype} + T_{qterm}^{op}$ for unary operations.
- User defined functions and data types can be statically eliminated (e.g. expressed in term of the standard QuestF-syntax) using the techniques presented in [Wim00].

The estimation of unary and binary QuestF-term-operations is accomplished in dependency to the characteristic of the chosen platform and hardware architecture. Some of them are listed below:

- *Size of registers and size of data types.* The proportion between these two factors influences the execution of assembler operations. If the maximum length of data types (in bytes) defined in the AutoFOCUS-system is greater than the capacity of the processor's registers, additional time-consuming arrangements must be taken.
- *Type of used assembler,* e.g. the number of operands (3/2/1/0) as well as the allowed addressing types (direct/indirect/stack pointer/index). These factors affect the execution time of operations and influence the number of needed read and write operations, which are discussed below.
- *The duration and number of read and write operations* are important factors for evaluation time. They correlate with the following specifics of the deployment architecture:
 - *Number and size of registers* affects the amount of write and read operations during the evaluation. In pure accumulator architectures, the intermediate results have to be written out to the memory every time they were calculated.
 - *The existence of caches and number of cache-levels* determines the duration of read and write processes. In the average case, they serve as a speed-up for these operations. However, in the worst case, the fetch operations can take long. I.e. if the needed value is not in the cache, the whole cache site has to be loaded. If the cache is full, another site has to be swapped out afore. For this reason many manufacturers of real-time embedded hardware/software systems don't use caching in their architectures.
 - *Optimizing compilers* are trying to reduce the amount of write/read operations by (semantic preserving) code transformations.
- *Existence and number of pipelines.* By using pipelines a certain degree of parallel execution is achieved by processors. This technique has also a worst case behaviour, which can be insufficient for real-time applications.

The presented certification approach expects a set of constant evaluation times for all primitive QuestF operations for every target platform. These constants serve as parameters of the estimation method presented in this section.

Literature

- [AutoFOCUS]** AutoFOCUS project homepage, dated June 23, 2003
<http://autofocus.in.tum.de>
- [BFP03a]** Botaschanjan, Fleischmann, Pister, "Integration of Classifications, Structuring and Process Models", Chapter 5, In: "Framework for Requirements", EMPRESS Compositum, 2003.
- [BFP03b]** Botaschanjan, Fleischmann, Pister, "Change Management of planned and unplanned Evolution", Section 5.1, In: "Method, notation, process for management of requirements in product families and across products", EMPRESS Compositum, 2003.
- [Bro98]** Manfred Broy, Informatik – Eine grundlegende Einführung, 2. Auflage, Springer Verlag, 1998.
- [Cla01]** Edmund Clarke *et al.*, "Model Checking", Third printing, The MIT Press, 2001.
- [Emp03]** Empress glossary, dated June 23, 2003
- [Fel03]** Emile Felix, Manual Development, "Integration and Test – DSW Platform, AHR-72-7DSW-SW MAN-0001", Jabil - company internal document, 2003
- [GZ01]** Thomas Gessner, Christian Zeidler, "Rule-driven component composition for embedded systems", In: "4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction", ICSE 2004
- [IEC61508]** Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems. International Electrotechnical Commission
- [ISO14977]** DS Danish Standards Association (Publisher), "DS/ISO/IEC 14977 Information Technology - Syntactic Metalanguage - Extended BNF", 1999-04-12, valid since 1998-10-01
- [ISO9126]** Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their use. International Organisation for Standardization, 1991
- [Kru00]** Philippe Kruchten, "The Rational Unified Process: An Introduction", 2nd Edition, Addison-Wesley, 2000
- [Mil89]** Robin Milner, Communication and Concurrency, Prentice Hall, 1989
- [QUG00]** Peter Braun, Heiko Lötzbeier, Oscar Slotosch: Quest User Guide. Version 1.0 of March 7, 2000.
<http://www4.in.tum.de/proj/quest/papers/UserGuide.pdf>
- [SW01]** J. Stafford, K. Wallnau: Is Third Party Certification Necessary? In:

Engineering, IEEE Computer Society Press, pp. 13-17, 2001

[Ter02] Rudi Stans, Process Architecture Smiley, October 2002, ITEA @Terminals project, (if99030);Jabil - company confidential document

[Wim00] Guido Wimmel, Specification Based Determination of Test Sequences in Embedded Systems, Master Thesis, TU München, dated February 15, 2000
http://www4.informatik.tu-muenchen.de/~wimmel/papers/Wim00_DA.ps