

Report from ASTEC–RT Auto project

Central Lock System Case Study

Tobias Annell
Department of Computer Systems
Uppsala University

Pontus Jansson
Automotive Systems, Gothenburg
Mecel AB

Abstract

Engineers today face increasing challenges due to complexity in their designs. To avoid costly design and debug iterations tool support for early validation and verification is needed. The traditional textual based specifications are non-formal and non-testable. This is why developers need tools for model based specification that supports early validation and verification.

In an attempt to promote technology transfer between industry and academia and to address the above mentioned challenges and need for tool support this paper presents an interface between a systems engineering tool, BUTLERTM⁽¹⁾ that is used by automobile manufacturers and their subcontractors and a formal verification tool, UPPAAL⁽²⁾.

The purpose of this interface and the method by which it should be used is to formally verify that the high level functional requirements of a design are met and not conflicting. Furthermore, the UPPAAL model provides early validation of specific features through simulation and design flaws are thus more likely to be avoided.

In order to evaluate the usefulness of the tool interface and the suggested method a case study of a real automotive application using both tools has been performed by the partners in conjunction with Saab Automotive whose part in the project has been to provide the functional requirements of a state-of-the-art vehicle central locking system in return for the verification results. The case study describes how high level functional requirements together with the functional architecture are exported from BUTLER to UPPAAL where behaviour is modelled formally.

The tool interface enables model exchange in a way that provides the engineer with sufficient support for formalisation of a high level functional description of a system in the UPPAAL tool. The tools have been used to create a model of rather high complexity that by simulation and verification has been shown to correspond to the functional requirements.

Keywords: Early validation and verification, formal verification, BUTLER, UPPAAL, functional modelling, model exchange.

1 Introduction

It is a fact that cars and other vehicles are becoming more and more complex with more electronics and embedded applications. Most (if not all) of these systems are critical to the functionality of the vehicle. Especially if we by critical consider the economical issues. Even a sound system in a car that does not function may cost the manufacturer considerable amounts of money. The traditional document/textual based development at the car manufacturers and their subcontractors today are changing into model based processes as the demands for formalism and tool support are necessary. Most of these embedded systems are also distributed over several computing nodes in the vehicle and connected through possibly several networks. Many of them also has real time constraints, such as engine control systems, automatic brake and anti spin systems, etc.

Model based early validation & verification

The inherent complexity of these systems calls for appropriate methods at an early stage during specification in order to verify that requirements are not conflicting. This case study suggests a

1 BUTLERTM is a registered trademark of Mecel AB, Sweden.

2 UPPAAL developed in collaboration between the Design and Analysis of Real-Time Systems group at Uppsala University, Sweden and Basic Research in Computer Science at Aalborg University, Denmark.

method where the high level functional requirements are used together with the functional architecture to create a behavioural model in a formal notation. The purpose of this behavioural model is to formally verify that the high level functional requirements are met and that they are not conflicting. Furthermore, the model provides early validation of specific features through simulation and the possibility to avoid design flaws by e.g. checking that no deadlocks are reached. The central locking system modelled in this case study is a generalised version of an actual Saab Automobile system. The case study input has been the actual functional requirements of the lock system. It is thus a purely logical description of the functionality and it does not take implementation aspects into account. Due to the complex nature of the combined functionality of a central lock system it is highly interesting to be able to formally verify the functional requirements.

Systems engineering with BUTLER

BUTLER is a systems engineering tool designed with this in mind – design and maintenance of system consistency for functional requirements, partitioning, functional and physical architecture and allocation of sub functions –. BUTLER is based upon a custom made database implemented in Microsoft Access for local development environments, or in Oracle for a distributed or multi-site environment. BUTLER is adapted to the development process that the customer has chosen for his distributed vehicle systems. A BUTLER system model is a hierarchically structured static description of an automotive system and it can be linked to tools suitable for modelling dynamic behaviour.

Modelling and verification with UPPAAL

Model-checking is a technique for automatic verification of computer systems, for example programs, protocols or specifications. It uses a model of the system, defined in a formal language, and uses an algorithm (implemented in a model-checking tool) to check whether the model has some property that is of importance. For example that a person cannot get locked inside a car.

In this case-study we have used the model-checking tool UPPAAL⁽¹⁾ [BLL+98]. The main feature of UPPAAL is that the tool can handle requirements containing real-time. It does this by representing the state space of the system symbolically so that not every time value has to be stored.

The UPPAAL tool include a graphical user interface containing an editor for the model, a simulator to see the model run and a verification interface where the requirements on the model can be specified. The verification server is separated from the user interface and can be executed remotely on a powerful machine in a network.

1.1 Related work

Previously UPPAAL has been used in several different case studies. The most common type has been verification of protocols of different kinds, such as communication protocols, see for example [HSL+97][DKRT97] or a longer list at the UPPAAL home page [ua01]. A case-study more similar to this is the study of the design of a gear controller in [LPW98]. The similarities lies in that the modelled system is a design rather than a protocol or an algorithm.

1.2 Outline

The rest of this paper is organized as follows: In section 2 the case-study is presented. In section 3 we present how we generate a skeleton of a model in Uppaal. Finally we present the formalization and the result of the verification in section 4.

2 Central locking system case-study

By “early validation & verification” we suggest a methodology in which the behavioural modelling of a system is performed at the very beginning of the development process, i.e. the early stages of requirement/system analysis. By formalising behaviour at an early stage the semantic gap between textual and formal requirements specification could be kept to a minimum. Furthermore, the formal model provides early validation of specific features through simulation and the possibility to avoid design flaws. Using the tools and the methodology outlined here the designer will be able to formally verify that the high level functional requirements are met and that they are not conflicting.

The export / import scenario that is considered covers export from BUTLER and import to UPPAAL. The systems engineer working on the design of a system using the BUTLER tool will thus be able to export the static design and functional requirements to UPPAAL where the behavioural modelling, simulation and verification can be performed. The result of the verification, i.e. the UPPAAL model can then be stored in BUTLER as a link to the generated files.

The BUTLER model expresses the high level functional requirements of the central lock system application and how they are decomposed, i.e. the functional decomposition into three hierarchical levels as illustrated in fig 1. Except for the structural information of the system, the textual requirements on dynamic behaviour is included as well, i.e. each model element has its own requirement attached to it.

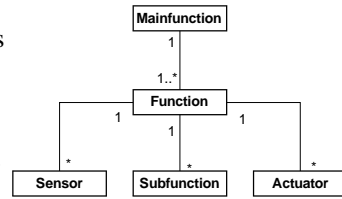


fig 1 A general functional decomposition

In fig 2 an overview of the lock system functional decomposition is shown down to the second hierarchical level for the entire

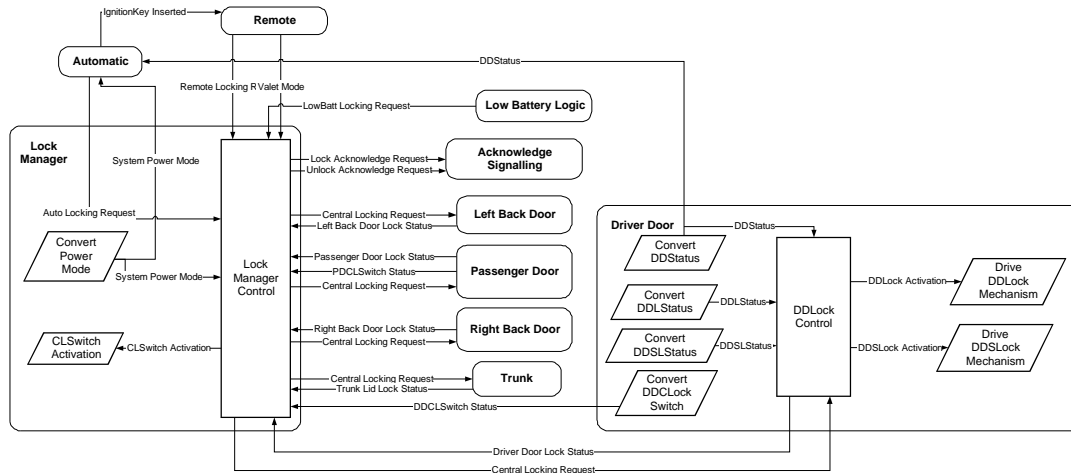


fig 2 Application overview

functionality. Selected parts like the Lock Manager and Driver Door functions have been detailed further and shows the lowest hierarchical level of sub-functions, logical sensors and actuators.

The platform consist of a typical car lock system with lock mechanisms, door knobs, remote control device and compartment switches. The system could be allocated onto a single ECU (Electronic Control Unit) or preferably a distributed system as illustrated in fig 3. The central lock system is thus highly distributed and the number of ECU's correspond to the number of mechanical system parts located in the doors, trunk, engine compartment and dashboard.

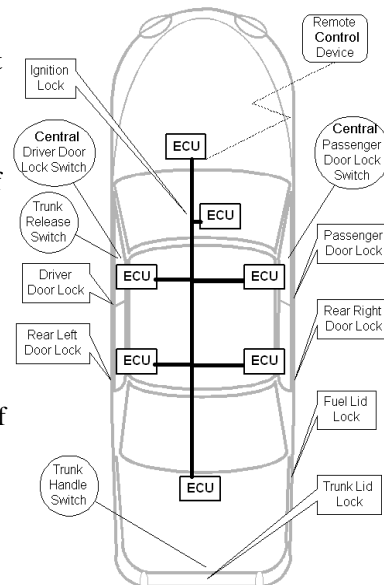


fig 3 Platform overview

2.1 Functionality

The central lock functionality is part of the vehicle's theft protection. The task of the central lock/unlock functions is to prevent unauthorised persons from getting access to the interior of the vehicle. The central lock functionality is closely related to other functions such as alarm, immobiliser, etc. The lock system modelled in the case study is a state-of-the-art central lock system according to Saab Automobile. Due to confidentiality issues this report only outlines some of the basic lock system functionality.

General functionality describes functions related to vehicle power modes, acknowledge signals and battery power levels. These functions could be; Acknowledgement of successful locking or unlocking provides audio-visual signals using turn indicators and the horn; Safety unlocking when the battery voltage reaches a level below 9V. **Automatic functionality** includes various lock / unlock functions that ensures a certain lock status depending on / triggered by timers, door status, vehicle speed or ignition key status. When the customer locks the vehicle with one or more doors open the system will not actually lock the doors until all doors have been closed for a certain amount of time. These functions motivate the use of timed automata for modelling and Uppaal for verification. **Compartment functionality** allows the driver to lock and unlock the doors, trunk and fuel lid from inside the car. The compartment functions are actuated by the driver and passenger door switches, the trunk unlock and fuel lid unlock switches.

Remote functionality allows the driver to lock and unlock the doors (and possibly actuate other functions) at some distance from the vehicle. This is accomplished by using the Remote Control Device (RCD) carried by the driver. The RCD has the following functions: Lock, Unlock, SECURITY–Lock, Trunk Unlock.

3 UPPAAL skeleton from the static structure

In this section we will describe how we generate a skeleton model from a static structure present in a tool like BUTLER. Our goal with the transformation is to keep the static structure so that it is easy to backtrack to the original description when the validation and verification of the behaviour has been performed. The information in the static structure that we may use as a base for the generated skeleton is mainly the decomposition into function blocks and the name and type of signals sent between them. The generated skeleton can be improved by adding some tags to the function blocks.

3.1 Design Guidelines

We introduce a few attributes to the BUTLER model description that will aid the creation of UPPAAL models. It is thus important for the BUTLER systems engineer to follow the guidelines concerning attributes below.

Active Sensors A sensor according to the BUTLER system description is a logical sensor and it could be considered either active or passive. When designing a system to be exported to UPPAAL it is necessary to add the *ActiveSensor* attribute to all sensors and set them accordingly. An active sensor produces a system input that triggers something, i.e. it generates an interrupt. This is typically a switch or a button that invokes a change of state, e.g. the central door lock switch in the driver or passenger door. A passive sensor provides sensory output that is polled when needed, e.g. a voltage level or a door status.

Actuator – Sensor pairs For actuators and sensors, an attribute is used to create a single process from each sensor and actuator pair that interacts with the same physical process. The attribute *PhysicalObjectPair* is defined as an unique name (char string) of a sensor–actuator pair. The driver door lock status for instance, is a consequence of how the lock motor is actuated, i.e. the “Convert DDLStatus” sensor that reads the status of the driver door lock mechanism reflects the input to the “Drive DDLock Mechanism” actuator that actuates the lock mechanism for the driver door. Both sensor and actuator in this pair is thus marked with the *PhysicalObjectPair* attribute “Driver Door Lock Mechanism”.

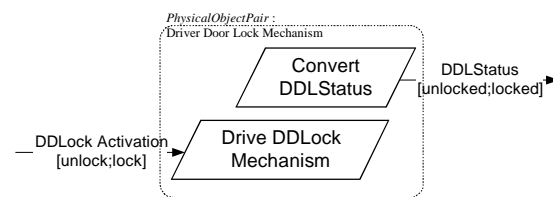


fig 4 Physical Object Pair

3.2 Generating a skeleton

A system in UPPAAL is defined with templates of processes³. These templates may have a list of parameters so that several slightly different processes can be instantiated. To define a system the instantiated processes that should be included are listed. Variables, clocks and channels may be defined as global on the system level. Each process may also define local variables and clocks. For further information on UPPAAL see for example[BLL+98].

In the context of this case study we consider three types of blocks namely functions, actuators and sensors and one abstract connector called signal.

The basic generation will produce one UPPAAL process for each function and variables and channels that will correspond to the signals in the BUTLER description. The hierarchical information in BUTLER, the functional decomposition, is used for grouping of processes and requirements in the UPPAAL import file. This results in an instantiated (flat) view of the system in UPPAAL, i.e. no templates are created. We will generate special constructs if the user has used the attributes described above to indicate special meaning to the element.

The signals to and from the function (process) is added to the templates parameter list. This basically means that the process should only use the variables and channels present in its parameter list. In other words we encourage the user to use only local variables within one process. But if the function,

3 Process is the UPPAAL term for an automaton

later in the modelling process, is divided into several UPPAAL processes this may no longer be true and can be abandoned.

3.2.1 Sensor – actuator pairing

Automatic generation of "physical object" automata, i.e. models of the environment from BUTLER Actuator/Sensor objects, is based on the *PhysicalObjectPair* attribute. The unique name of the attribute is used as the name of the combined process in the generated UPPAAL process.

Based on the input and output signals to a function it is possible to create a stub automata for the UPPAAL process. Two common constructs are identified in the following cases.

2 in – 2 out This can also be called a lock. If we have a two valued signal to the actuator in a pair for setting the state of the mechanism [lock; unlock] and a two valued signal from the sensor reading the state [locked; unlocked] as illustrated in fig 4. We can create an automaton like in fig 5.

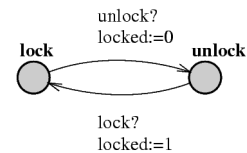


fig 5 Lock automaton

2 in – 1 out The *ActiveSensor* attribute enables us to generate automata with either channels (active sensors) or variables (passive sensors). Active sensors are instantaneous (interrupts) while passive are levels (polled). If we have a two valued (Boolean) active sensor e.g. a switch [pressed; NOT pressed] that opens the trunk lid and a two valued signal to the actuator, in this case the switch which can be [enabled; disabled], we can create an automaton like in fig 6.

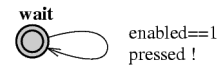


fig 6 Active sensor

3.2.2 Signals

The signals in the BUTLER description can be automatically created as a pair of a global variable and a channel in the UPPAAL model. The value to be passed is first set in the variable and then the channel is activated. We classify signals according to the following:

Signals from sensor blocks to function blocks Signals which are two valued, e.g. signals with the *2waySwitchStatus* attribute, can be created without a global variable. Sensor signals with more possible values (enumerated) such as the *3waySwitchStatus* must be created with a global variable.

Signals from function blocks to actuator blocks Typically each value in the signal is associated with one channel. No variable is created. Modifications to this is possible, the alternative is that the actuation is only a writing to a global variable. This is for example the case in the *2 in – 1 out* case above.

3.2.3 Global variable definitions

In the variable section we will automatically generate variables used by the communication between the processes. All the signals are represented by variables and/or channels.

Signals with numerable types are given constants for each value. And the variable will be a limited integer variable. A three valued numerable signal will generate this:

```

const VAL1 0;
const VAL2 1;
const VAL3 2;
int[0,2] SignalName;
chan SignalNameCreated;
  
```

All integer variables with limited range (including Boolean) should use the range declaration:

```

int[minrange, maxrange] varname;
  
```

A signal that is sent to many different functions or actuators must be created in several instances, since UPPAAL does not support multi-cast communication. In the example above, the variable and channel declaration must be duplicated and modified as:

```

int[0,2] Target1_SignalName;
chan Target1_SignalNameCreated;
int[0,2] Target2_SignalName;
chan Target2_SignalNameCreated;
  
```

3.2.4 Process declarations

A process is created for every function and actuator–sensor pair and unpaired sensor. The names are the same as for the function, pair or sensor. In the parameter list the signal names and the variables (if any) associated with the signal is included.

A function sending a three valued signal to two different targets will include (at least) this in its parameter list:

```
process FunctionName(int[0,2] Target1_SignalName, Target2_SignalName; chan
Target1_SignalNameCreated, Target2_SignalNameCreated){}
```

Pseudo–code for generation of UPPAAL model

Pseudo code to generate a skeleton system in UPPAAL that correspond to the structure in BUTLER is included in the appendix. The generation is a one way creation of an UPPAAL XML document from a BUTLER XML document.

The transformation is presented in a way inspired by the Visitor pattern. For each element in the BUTLER XML document we either create something in the UPPAAL document or we visit a child node in the BUTLER XML–tree. Some elements in the BUTLER document cannot be used to generate the UPPAAL model and are omitted. The tags that refers to the BUTLER document are prefixed with bu: and those with UPPAAL are ua: in the XML name–space fashion.

4 Behavioural modelling in UPPAAL

With the help of the generated skeleton structure based on the principles outlined in the previous section we have used UPPAAL to construct a model of the lock systems behaviour.

In UPPAAL models are described with timed automata [AD94]. Timed automata are finite state automata extended with integer variables and real valued clocks. UPPAAL can handle a couple of further extensions to timed automata, such as committed and urgent location, of which most are included in the input language to improve the efficiency in verification. In particular proper use of committed locations will reduce the state space of the system considerably.

While modelling the functionality the original static structure have been modified in different ways. For example functions such as the automatic ones have been split up into separate processes and others have been composed together. The resulting model of the locking system consists, as a core, of at least 22 separate automata. This core include the functionality in the Lock Manager, Automatic, Remote, Passenger Door and Trunk function blocks, see fig 2. It does not include the back doors or the battery logic. This reduces the size of the state–space when the verification is performed. Depending on what properties we are interested in we may also need to include observer automata. Table 1 in the appendix present an overview of the processes included in the model.

Some parts of the functionality does include timing requirements. In particular the automatic functionalities are specified with requirements on action within certain deadlines.

In the model we have also added some processes that models the environment of the system. These are of course needed to provide the system with inputs that is needed to “drive” the system when it is verified. We have aimed at keeping the environment and the control as separated as possible, the environment should be the source for non–determinism and the control should be deterministic. This has not always been possible to maintain when we have been forced to optimise the model to fit into memory when verifying. The environmental features could still be considered to be situated at the “borders” of the model. In table 1 the processes including environmental features are indicated with an “env” superscript.

In figure 7 to 12 we show, as an example, the automata that make up the functionality in the Driver Door function block. If we compare with the application overview in figure 1 the automaton in figure 7 correspond to the *DDL Lock Control* sub–function, the automaton in figure 11 correspond to the sensor *Convert DDSL Status* and actuator *Drive DDL Lock Mechanism*, the automaton in figure 10 correspond to *Convert DDSL Status* and *Drive DDSL Lock Mechanism*, the automata in figure 8 correspond to *Convert DDCLock Switch*. The automata in figure 9 and 12 models the environment and has no representation in the BUTLER system. Note that these automata does not include any timing.

5 Conclusions

This work has been focused on investigating how tools and techniques from academia can be used in a model based approach in industrial applications. To do this we propose, and have used, a methodology consisting of using a system engineering (CASE) tool to specify the static structure and functional decomposition and to export a skeleton system for a tool where the behaviour can be modelled and verified.

As a part of the work we have verified the correctness of a central lock system provided by Saab Automotive. The UPPAAL model of the functional behaviour clearly illustrates the high level of complexity in a system of this type. The model consists of 22 (or more) parallel communicating automata that each provides some functionality to the system. The work has resulted in a specification that has a considerably higher degree of completeness due to elimination of ambiguities.

Experiences drawn from using UPPAAL in this case study tell us that this type of tool is not ready for use by the average engineer yet. Considerable amounts of in-depth knowledge about the verification tool and formalism is needed and it can not be considered a "push-button" technology. An expert in between the system/application engineer and the formal model is needed to use the tool in a proper way. However, if one considers the roles normally found in the automotive industry between manufacturers and subcontractors the requirements could be specified and structured with a CASE-tool. Then, with the possibility to export the structure to a tool that another person (the subcontractor/expert) will use to perform the verification, we believe that the methodology we suggest is a useful one.

References

- BLL+98: Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Wang Yi and Carsten Weise, New Generation of UPPAAL., In proceedings of the International Workshop on Software Tools for Technology Transfer, 1998
- LPW98: Magnus Lindahl, Paul Pettersson and Wang Yi, Formal Design and Analysis of a Gear Controller, In Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems., 1998
- ua01: Uppaal homepage, 2001, <http://www.uppaal.com>
- DKRT97: P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans, The bounded retransmission protocol must be on time!, In Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Enschede, The Netherlands, April 1997. LNCS 1217, pages 416-431., 1997
- HSL+97: Klaus Havelund, Arne Skou, Kim G. Larsen and Kristian Lund, Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL, In Proceedings of the 18th IEEE Real-Time Systems Symposium, 1997
- AD94: R. Alur and D.Dill, Automata for Modelling Real-Time Systems., 1994

Appendix

Pseudo-code for generation of UPPAAL model

<pre><bu:mainfunc> create new Uppaal XML documant named <bu:name>.xml visit all children /* <bu:req>, <bu:func> */</pre>
<pre><bu:func> insert new <ua:template> with name as <bu:func> visit all children /* <bu:req>, <bu:output>, <bu:input>, <bu:sensor>, <bu:actuator>, <bu:subfunc> */</pre>
<pre><bu:subfunc> insert new <ua:template> with name as <bu:subfunc> visit all children /* <bu:req>, <bu:output>, <bu:input> */</pre>

<pre> <bu:req> write a new query in .q-file with <bu:name> and <bu:description> as comment </pre>
<pre> <bu:input> /* we assume that the signals are defined where they are produced, so we do not need to define any new global variables here. */ for each <bu:signal> add channel to templates parameter list add the variable to the templates parameter list </pre>
<pre> <bu:output> visit all <bu:signal> if <bu:attrib>:s datatype is enumeration create constants in global variable section for each enumeration value </pre>
<pre> <bu:sensor> if <bu:attrib> with <bu:name> "PhysicalObjectPair" find the matching <bu:actuator> create <ua:template> with <bu:sensor>:s <bu:name> /* 2 special cases */ if <bu:sensor>:s value range == 2 and <bu:attrib> with <bu:name> "ActiveSensor" and <bu:actuator> in matching pair has value range == 2 create "2 in - 1 out" type automata in <ua:template> else if <bu:sensor>:s value range == 2 and <bu:actuator> in matching pair has value range == 2 create "2 in - 2 out" type automata in <ua:template> else /* normal case */ for the <bu:actuator>:s <bu:signal> create channel in global variable section add the channel to the <ua:template>:s parameter list visit <bu:sensor>:s <bu:signal> else create <ua:template> with <bu:sensor>:s <bu:name> visit <bu:signal> </pre>
<pre> <bu:signal> /* Output signal (from function or sensor) */ if <bu:attrib>:s datatype is enumeration create constants in global variable section for each enumeration value create enumerable, integer or boolean variable in variable section add variable to templates parameter list for each <bu:func>, <bu:subfunc> or <bu:actuator> that the signal is sent to /* we need this to emulate multi-cast */ create channel in global variables section add channel to templates parameter list </pre>
<pre> <bu:actuator> if not <bu:attrib> with <bu:name> "PhysicalObjectPair" create empty <ua:template> with <bu:sensor>:s <bu:name> </pre>

Summary of UPPAAL models

Function	Processes (subfunction)	Locations
Driver door and lock	Driver Door ^{env}	2
	Driver Door Lock	3 (1)
	Driver Door Security Lock	2
	Driver Door Lock Control	4 (2)
Passenger door and lock	Passenger Door ^{env}	2
	Passenger Door Lock	3
	Passenger Door Security Lock	2
	Passenger Door Lock Control	4 (2)
Trunk	Trunk	3 (1)
	Trunk Handle ^{env}	1
	TrunkRelSwitch ^{env}	1
	TrunkControl	2 (1)
Remote control device	RemoteFob ^{env}	3
	ValetFob ^{env}	3
	OpenWith Key ^{env}	1
	IgnitionKey	3 (1)
Central	Central Locking Manager	24 (12) (no Back Door, no Battery)
Compartment	Central Door Lock Switch ^{env}	1
Automatic	AutoCarRelock	4 (2)
	AutoTrunkLock	2
	AutoTrunkRelock	3 (1)
	SpeedSensor ^{env}	2

Table 1 Overview of processes in the locking system model. The env superscript indicates that the process contains environmental features. The size is given as locations (the number in parenthesis is the number thereof that are committed locations).

Document Type Definitions – DTD's

UPPAAL DTD

```

<!ELEMENT nta (imports?, declaration?, template+, instantiation?, system)>
<!ELEMENT imports (#PCDATA)>
<!ELEMENT declaration (#PCDATA)>
<!ELEMENT template (name, parameter?, declaration?, location*, init?,
transition*)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name x CDATA #IMPLIED
y CDATA #IMPLIED>
<!ELEMENT parameter (#PCDATA)>
<!ATTLIST parameter x CDATA #IMPLIED
y CDATA #IMPLIED>
<!ELEMENT location (name?, label*, urgent?, committed?)>
<!ATTLIST location id ID #REQUIRED
x CDATA #IMPLIED
y CDATA #IMPLIED>
<!ELEMENT init EMPTY>
<!ATTLIST init ref IDREF #IMPLIED>
<!ELEMENT urgent EMPTY>
<!ELEMENT committed EMPTY>
<!ELEMENT transition (source, target, label*, nail*)>
<!ATTLIST transition x CDATA #IMPLIED
y CDATA #IMPLIED>
<!ELEMENT source EMPTY>
<!ATTLIST source ref IDREF #REQUIRED>
<!ELEMENT target EMPTY>
<!ATTLIST target ref IDREF #REQUIRED>
<!ELEMENT label (#PCDATA)>
<!ATTLIST label kind CDATA #REQUIRED
x CDATA #IMPLIED
y CDATA #IMPLIED>

```

```

<!ELEMENT nail EMPTY>
<!ATTLIST nail x    CDATA #REQUIRED
              y    CDATA #REQUIRED>
<!ELEMENT instantiation (#PCDATA)>
<!ELEMENT system (#PCDATA)>

```

BUTLER DTD

```

<!ELEMENT mainfunc (name, description?, req*, func+)>
<!-- A mainfunction has a name, an optional description and zero or more
requirements. A mainfunction contains at least one function. !-->
<!ELEMENT func (name, description?, req*, input*, sensor*, actuator*,
subfunc*, output*)>
<!-- A function has a name, an optional description and zero or more
requirements, inputs and outputs. The function is an encapsulation of
subfunctions, sensors and actuators. !-->
<!ELEMENT subfunc (name, description?, req*, input*, output*)>
<!-- A subfunction has a name, an optional description and zero or more
requirements, inputs and outputs.!-->
<!ELEMENT sensor (name, description?, req*, signal, attrib*)>
<!-- A sensor has a name, an optional description, zero or more
requirements and attributes and ONE output signal. !-->
<!ELEMENT actuator (name, description?, req*, signal, attrib*)>
<!-- An actuator has a name, an optional description, zero or more
requirements and attributes and one input signal!-->
<!ELEMENT req (name, description?, attrib*)>
<!ATTLIST req class (functional|non-
functional|environmental|performance)#IMPLIED>
<!-- A requirement has a name, an optional description, zero or more
attributes and an optional class!-->
<!ELEMENT signal ((name, description?, attrib+)|ref)>
<!ATTLIST signal id ID #REQUIRED>
<!-- A signal has a name and id, an optional description and at least one
attribute. Alternatively a reference to a signal id !-->
<!ELEMENT attrib (name, description?)>
<!ATTLIST attrib datatype
(boolean|integer|float|string|enumeration)#REQUIRED
enum_items CDATA #IMPLIED
value CDATA #IMPLIED
default CDATA #IMPLIED>
<!-- An "attrib" has a name, a datatype attribute and an optional
description. !-->
<!ELEMENT output (signal+)>
<!ELEMENT input (signal+)>
<!ELEMENT name (#CDATA)>
<!ELEMENT description (#CDATA)>
<!ELEMENT ref (#ID)>

```