Abstract: Complex automotive systems are composed of subsystems and components in a deep hierarchy, often designed by different development partners or reused from preexisting projects. It is therefore a challenging task to break down requirements into sub-requirements fitting the scope of the subsystems and to simultaneously demonstrate that the integrated system fulfills both functional and safety requirements specified on the top-level. Contract-based development is a popular approach for breaking down requirements onto components by means of assumptions and guarantees. However, most current approaches are based on a formal semantics and therefore limited in their expressive power and their acceptance by practitioners from automotive industries. We propose a semi-formal approach that allows specifying assumptions and guarantees at component interfaces in a language with well-defined syntax, but leaving the verification of fulfillment of the contract by a component to expert decision. However, some of the relevant refinement relations can be formalized and automatically checked. We describe our prototypical Eclipse tool that allows the annotation of components with assumptions and guarantees, and the partial checking of the decomposition. We show the applicability by a case study of an automotive electric drive system.

Keywords: contracts, safety, functional safety, requirements engineering, assumption, guarantee.

1 Introduction

Automotive systems have kept growing in complexity for the past decades, and most automotive embedded systems are safety-related systems. Vehicle functions, for example, in the domains of hybrid/electric powertrain or advanced driver assistance systems (ADAS) are distributed across many electronic control units (ECUs), each of them contributing to many different functions. These ECUs again consist of a complex network of components implemented in different technologies. ECUs or components are usually designed by different departments or external suppliers. Some of them are reused from past projects or have originally been developed stand-alone for later usage in various vehicle projects by different OEMs (“Safety Elements out of Context” in ISO 26262 language). Often we noticed a lack of understanding which requirements and properties a specific functional unit supposedly fulfills, and under which assumptions it had originally been developed. This makes it difficult for the vehicle OEM to provide a sound safety argument for the integrated system. As the degree of automation in vehicle

1 Berner & Mattner Systemtechnik GmbH, Gutenbergstr. 15, Berlin, bernhard.kaiser@berner-mattner.com
2 Alpha EOS, Stuttgart, s.sonski@gmail.com
3 Berner & Mattner Systemtechnik GmbH, Gutenbergstr. 15, Berlin, suryo.buono@berner-mattner.com
4 Freie Universität Berlin, Computer Science Institute, Takustr. 9, 14195 Berlin, hauke.petersen@fu-berlin.de
5 Berner & Mattner Systemtechnik GmbH, Gutenbergstr. 15, Berlin, justyna.zander@berner-mattner.com

1595
constantly grows, functional safety gets more and more intertwined with the correctness, accuracy and performance of the nominal function, so the above considerations are not restricted to safety in a narrow sense but apply to correct development of component-based automotive systems in general. Accordingly, system architects need help with systematically decomposing requirements onto components, evaluating candidate components w.r.t. given requirements and demonstrating that all parts fit each other and the integrated system is proper for the selected purpose. Given the situation in industry today, it may be postulated that even an argument that is not provable by a computer but needs manual review is better than none. The decomposition of properties like value accuracies, time bounds for system reactions or safety integrity levels still requires a lot of scientific work and we therefore doubt that a formal solution in combination with a user friendly interface for industry practitioners will be available in the next few years.

An increasingly popular approach for decomposing requirements onto components and showing their compatibility is the development process by contracts. Contracts support partition of work and isolate properties at interfaces. Contracts have first been proposed for the compositional verification of linear software programs [Me92] but were in later years extended by a large research community towards component-based software, i.e. software components as independently created entities that communicate with other components via interfaces. Today, contracts are often understood as sets of assumptions, on which a component relies to be fulfilled by its environment, and guarantees a component makes towards its respective environment. Both assumptions and guarantees refer to observable behavior at the component interfaces. Often, formal notations are used to express contracts (i.e. some forms of predicate logic, temporal logic or other types of modal and higher-order logic), allowing for refinement correctness check of contracts down the architectural hierarchy.

Although the expressiveness of formal contracts has increased impressively, many approaches are still on the level of academic research and lack direct applicability to automotive industries for several reasons: first, when aiming at a formal underpinning, it cannot be expected that the expressive power of today’s contract annotation languages will soon fit the great variety of important properties of component-based automotive systems, encompassing properties such as correct function (in whatever sense), accuracy timing and safety integrity. In particular, the latter is rather a meta-property that cannot be formally proven but needs expert judgment for verification. Second, from their origin in informatics, many of the formalisms are only applicable to software components, not system level components. For example, many of the existing frameworks for contracts are based on discrete–time and/or discrete-state components, whereas the physical world is mainly characterized by continuous-valued systems. Last, automotive systems are designed in cooperation of engineers from different disciplines ranging from software engineering over electronics to mechanical engineering, many of them still reluctant to learn and apply formal languages from computer science, but accepting programming-language style notations or graphical executable notations such as Simulink.

Therefore we came to the idea to try it the opposite way: believing that assumptions and
guarantees are a valuable aid for stating, decomposing and allocating requirements, even if written down in natural language and checked manually, we propose a semi-formal notation for assumptions and guarantees. Leaving aside formal semantics in the beginning, we start by proposing a structured language, checkable for well-formedness and yet acceptable to practitioners. Starting from there, we can go on with adding formal semantics at least for some aspects and provide checkers for these aspects. For instance, if we manage to provide a compare function that is able to conclude that an accuracy of 5% is better than 10% or that ASIL B is better than ASIL A, then the validity of an assignment of requirements to a set of given components (often referred to as dominance relation) can be confirmed, even without knowing how properties like ASIL or accuracy could be formally defined. The compliance of an individual component with the guarantees still needs to be verified manually, by manual review, simulation, or testing. The approach works in both cases: the breakdown of requirements onto components to be developed, and the reuse of pre-existing components in new projects, provided that they had been specified by contracts in the past. Still the way is open to add formal semantics for more and more aspects in the future (e.g. the validity of an ASIL decomposition can be checked if the ASIL decomposition rules from ISO 26262 are added, or the achievement of some timing or accuracy bounds can be checked by model checkers or simulation, if the semi-formal notation is enriched by a formal underpinning for certain properties). Therefore, we do not see our proposal as a competitor to formal approaches, but rather as a complementary to them, preparing the ground for their acceptance in industry.

In a master thesis at Berner&Mattner Systemtechnik [So13] the approach is exemplified by using a tentative specification language which resembles commonly known object-oriented (OO) programming languages like C++. As a proof of concept, a corresponding prototypical Eclipse-based modeling and checking tool was developed. We applied the approach to a case study of an electric drive that is a simplified but yet realistic abstraction of electric powertrain drive systems we encounter in our day-to-day work with well-known German vehicle OEMs and suppliers. This paper summarizes and explains the experiences we gained with our semi-formal approach.

The rest of the article is structured as follows: Section 2 provides a brief overview on contracts; Section 3 presents our approach and introduces the annotation language for assumptions and guarantees. Section 4 introduces the prototypical tool, Section 5 gives an insight to our case study of an automotive electric traction drive and Section 6 concludes with an outlook on the next steps on the research agenda.

2 Background and State-of-the-Art

2.1 Contract-Based Development in the Domain of Software Engineering

Contracts as a mean to assure correctness of computer programs have initially been
proposed by Bertrand Meyer [Me92]. The idea is to avoid errors in the collaboration of
different (sub-) functions of a software program by making the expectations of a
subfunction towards its caller and vice versa explicit (e.g. regarding the range of allowed
values that some variable may take). Thereby, caller and callee engage in a sort of
formal agreement, called contract. The expectations to be fulfilled at the entry point of a
subfunction are called preconditions and the expectations at the exit point are called
postconditions. Preconditions and postconditions may be statements that evaluate to true
or false (e.g. parameter x is greater or equal than zero), which can be stated in a formal
language which is intrinsic part of the programming language or available as an add-on
(e.g. assert-Macros in C language, Object Constraint Language OCL in UML). The
expectation of the architect is that at runtime the expression always evaluates to true
when reaching the given point in the program flow.

2.2 Contracts for Component-Based Development of Embedded Systems

Contracts in this original sense are applicable to software programs written in linear
programming languages. Software and systems architecture for technical systems,
however, is today designed and structured using component-based modelling techniques,
such as UML Component Diagrams, SysML Internal Block Diagrams, or the data flow
oriented and hierarchical subsystems of Simulink. Software and hardware components
may be delivered from different suppliers, purchased as standard components off-the-
shelf, or reused from former projects. They are integrated by system suppliers or the
OEM (e.g. vehicle manufacturer). OEM releases the embedded system to the market and
takes responsibility for the correctness, reliability, and safety of the entire system. It is
reasonable to transfer the contract concept to this setting.

Contracts described in related work (for an overview see [Be12]) mostly use the concept
of assumptions and guarantees. The contract is fulfilled, if, for the integrated system,
every assumption is implied by an appropriate guarantee by some peer component.
There have been proposals for the usage of contracts to specify real-time properties of
continuous-valued controller structures and the control error of such systems (e.g.
[Sc14]). Many times, contracts have been proposed for the functional safety domain
([BHM03][Fe07][Ba10] and others). Contracts have been applied to UML/SysML
models as well as Simulink models [Bo11] that play an important role in automotive
engineering.

3 Approach to Annotate Components with Interface Contracts

Our approach, first developed in [So13] is built upon Interface Contracts, which occur
between neighbor components. Their usage for the recursive breakdown of system
requirements onto the system’s components has been elaborated in [BKW15] (cf.
Section 3.3). Interface Contracts are bound to the input and output interfaces (ports) of a
component and are engaged between neighbour components that are directly linked by
signal flows (producer and consumer of some signal). In our approach, we assume the
system architect to be an omniscient observer. Therefore, it is allowed, for instance, to describe the accuracy of some measurement signal as the maximum expected deviation (e.g. 6 sigma) to its real-world counterpart signal, or the delay of some internal event as being the time passing between some event in the real world (e.g. some vehicle crash occurs) to the internal event (e.g. the command to some airbag ignition unit to inflate).

Our proposed process of requirements refinement and allocation by contracts comprises several steps: first, requirements are specified on system level. These form the guarantees to be fulfilled by the system under development, provided that the assumptions about the system’s operational environment hold. Next, the architect proposes a candidate system architecture in terms of components (here understood as functional blocks, independent of their realisation in hardware or software) linked by signal flows via their interfaces, with the goal in mind that this architecture will fulfil the requirements of the whole system. This process can be repeated recursively. The overall task described by the system requirement is decomposed into sub-tasks annotated to the components (e.g. “Switch off power-stage when motor current gets higher than x within y ms [ASIL D]” → C₁: “Measure motor current”; C₂: “Compare Current measurement to x and trigger switch-off command if higher”; C₃: “Switch off power-stage on receiving switch-off command”). At each port, signal properties like accuracy, delay or safety integrity level are specified in an assertion language that serves for both assumptions and guarantees. A component that consumes a signal at its input can rely on the specified property as an assumption. A component that provides some signal at its output must guarantee the specified property so that this guarantee becomes a requirement for this component. The architect has to negotiate acceptable assumptions and guarantees with component suppliers, or look for readily available components that fulfil the required guarantees under the given assumptions. Where input/output ports of components are directly connected to input/output components of the surrounding system, assumptions and guarantees propagate one level up. Verification is done by checking if every assumption is satisfied by a guarantee (referring to a connected pair of output port and input port) that is at least as strong as supposed by the assumption. An example of the usage of Interface Contacts is shown in Figure 1:

<table>
<thead>
<tr>
<th>Contract</th>
<th>Assumption</th>
<th>Guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>G: output.imprecision&lt;1% output.integrity=ASIL_B</td>
<td>A: input.imprecision&lt;2% input.integrity&gt;=ASIL_A</td>
<td></td>
</tr>
<tr>
<td>G: input.imprecision&lt;10% input.integrity&gt;=ASIL_B</td>
<td>G: output.imprecision&lt;5% output.integrity=ASIL_B</td>
<td></td>
</tr>
</tbody>
</table>
The component under consideration is $C_i$. It consumes an input value from its predecessor in the signal chain, $C_{i-1}$, and provides an output signal to its successor $C_{i+1}$. It assumes certain quality properties regarding the input signal, e.g. an imprecision (with respect to the true value in the real world) of less than 2% and a safety integrity of at least ASIL A (according ISO 26262 scale). The producer $C_{i-1}$ of the signal guarantees an imprecision of less than 1% for the same signal, which is better than assumed, and integrity of ASIL B. Therefore the contract between these two components is valid and the components are compatible to each other.

3.1 Syntax of the proposed Contract Annotation Language

To automate the described concept by a software tool, it is necessary to define a syntax (definition of well-formedness off the semi-formal notation) for assertions (assumptions or guarantees), even if the semantics of terms like “imprecision” or “ASIL” is not defined and must be interpreted by humans. The notation should be acceptably understandable to domain engineers. For first experiments, we invented a contract annotation language that has a high degree of similarity with well-known OO programming languages such as Java or C++, and that allows expressing three types of relevant properties: delay, accuracy and automotive safety integrity level (ASIL). An example of the syntax structure is given below:

```
componentName.portName.propertyName relational_operator value [unit]
```

Listing 1: Syntax example of the proposed contract notation language

The assumptions and guarantees are mapped to elements of the modeled system by their unique names. The `componentName` maps them to a unique component of the system, while the `portName` maps the given element to a precise signal port (input or output) of that component. Signals are logically typed as event signals, enumerative signals or continuous signals. These can be compared to reference values, either as binary comparison or by comparing their relation to a given value (equal, greater, or less than the value), which is a precondition for dominance checking. The annotation language allows some predefined ordinal scale values such as ASIL_A to ASIL_D and the optional use of a set of relative or physical units for continuous values as %, ms or V in order to prevent user errors by confusion. The same syntax applies to assumptions and guarantees; the actual role of an element is deducted by its placement. The prototypical tool provides syntax checking for the assumptions and guarantees typed in by the user.

3.2 Proposed Workflow for computer-aided checking

The proposed workflow for the computer-aided checking of the contracts, as
Lightweight Contracts for Safety-Critical Automotive Systems

implemented in the prototypical tool in Section 4, is as follows: (i) the system is graphically modeled using components and their connecting signals, (ii) components are annotated with their assumptions and guarantees directly in the model using the proposed annotation language, (iii) the analysis algorithm is run, doing an automated check for completeness (there are no assumptions without matching guarantees) and consistency (each guarantee implies the assumption it is linked to) of the contracts, (iv) the results are visualized and can be inspected. Following the workflow of the development processes of the embedded system, this sequence can be repeatedly carried out in an iterative fashion.

3.3 Contract Checking Algorithm

To perform the mentioned type of contract checking, an algorithm has been designed and implemented in the prototypical tool. The concept of this algorithm works as follows. First, all components are collected and saved in a list. All children of a component are added to the list of components recursively. For all assumptions, corresponding guarantees are searched, which are selected by the fact that they refer to the same signal (i.e. the one that connects output of one component to input of another) and property (i.e. imprecision or time delay or safety integrity level). If a guarantee is found, it is entered into the same contract dataset where the assumption came from. If no suitable guarantee is found, the assumption is checked for transferred assumptions, i.e. assumptions that have their origin on the next higher level of hierarchy. In case a guarantee was found and the units of guarantee and assumptions are equal, the values are compared with the relations defined in the guarantee and assumption. If the guarantee provides a better or equal signal quality than claimed by the assumption (e.g. assumption: Safety Integrity Level of ASIL A, guarantee: Safety Integrity Level of ASIL B), the contract is marked with an OK label, otherwise it is marked as NOK (not ok). If no guarantee is found, the contract is set to MISSING, which indicates, that no appropriate guarantee for the assumption was found. Additionally the guarantees are also checked for transferred guarantee compliance. To solve the problem of the verification of transferred statements, assumptions and guarantees have the same super-class so that both can be used in the position of an assumption or a guarantee in the contract.

4 Prototypical Tool

For evaluation of the idea a prototypical tool was created. The demonstrator includes the necessary functionality for modeling a system in a block diagram (actually a simplified form of a SysML Internal Block Diagram), annotating assumptions and guarantees for a set of relevant properties, automatically matching and checking contracts using the proposed algorithm and visualizing the results in a table. The user interface is based on the widespread Eclipse platform with its Rich Client Platform extension facility. The graphical editor is based on the Graphiti framework, which allows a quick prototyping of almost any kind of graphical representation based on a meta-model of the items to be modelled using the Eclipse Modeling Framework (EMF) [St08]. The proposed user
interface is based on two main views, illustrated in Figure 2: (i) the editor view showing the graphical representation of the system under construction and (ii) a table based view showing the results after analyzing the system contracts. In addition, it contains some supporting views and palettes for quicker access to functions and projects.

The editor displays a graphical model of the static architecture of the system under development (SysML Internal Block Diagram). Rectangles denote components, small squares with right or left pointing arrows on the outer borders of a component denote output and input ports, respectively, and lines connecting these ports denote connections of the ports via signals. The hierarchical nature of the modeled system is denoted by components that graphically contain other components. The model elements can be added by dragging them from a supporting palette view. In a later professional version, the model would be importable from existing UML/SysML modeling tools. Also, an import of block models from Simulink would be desirable; we have already implemented this feature in another related master thesis [Ra14].

After creating a system model, the system architect annotates assumptions and guarantees. These are added using the annotation language defined in Section 3.1. The annotated assumptions and guarantees are visualized as elements inside a component.

![Figure 2: Screenshot of the tool GUI](image)

The contract check engine implements the algorithm explained in Section 3.3 and can manually be started from the tool GUI. The results are then displayed in the tabular Result View. The table lists each detected contract with the following details in distinct cells of the table: The component that declared the assumption and the component that declared the guarantee, the guarantee (G) and the assumption (A) itself in the syntax explained above, and its type (G or A). A field “Aggregation” shows the result of the calculation of a guarantee and its referenced assumption by using the declared operation. A last column shows the verification results (OK, NOK and MISSING). By colors, the table clearly marks the remaining issues (contracts that do not match and assumptions with missing guarantees). Thereby, the correctness of the intended system architecture or the aptitudes of a reused component for its new context become visible at a glance.
5 An Automotive Electric Drive Case Study

The method and the tool have been evaluated based on the example of a simplified automotive electric traction drive system, as first introduced in [Ka12]. The system consists of a three-phase synchronous machine (PSM) and an inverter with 6 IGBT transistors as power switches, along with the required driver circuitry. The power stage is commanded by PWM (pulse width modulation) signals generated by a microcontroller, on which a complex software system is running. Among many other tasks, the software performs the closed-loop control algorithm (field-oriented control) to control the torque of the PSM by modulating a voltage system. The torque reference value is delivered via a data bus and represents the value derived from the accelerator pedal angle. The actual values of a number of physical quantities required for control (e.g. phase currents or rotor angle) are measured by suitable sensors with corresponding analog signal conditioning circuitry and then converted to digital numbers. This example is sufficiently similar to real vehicle drive systems, and for such systems, a set of safety goals is commonly known (we have meanwhile implemented an operating electric drive controller for a 1:8 scale toy car on this base, and have started developing a technical safety concept for it [Fe14], which is inspired from actual industry projects). A typical example of a safety goal is to prevent the machine from applying unauthorized or excessive torque, which could lead to an uncontrollable situation and in consequence to a road accident. Breaking down this requirement stepwise from vehicle level to controller level using our approach, one resulting technical safety requirements could, for instance, read as “It shall be detected if a phase current is higher than the commanded current by more than 10A, and if so, the power stage shall be disabled within at maximum 200 ms. [ASIL D]”. During system refinement and many iterations involving different safety analyses (such as FMEA or Fault Tree Analysis), safety mechanisms (e.g. signal plausibility checks) are added to the system, and the initial safety goals are transformed to more specific requirements, e.g. to detect that a phase current measurement value is not trustworthy, and to do so within a given time and with a defined accuracy. The example presented here starts at that point and deals with some of those checks.

For applying our semi-formal contract approach to this system, the first step is to model its static architecture. Each block and each signal is annotated with a verbal description of what it does or means (e.g. “compares phase current values and generates shut-off command when they seem no longer to be in a plausible correlation”). Figure 3 illustrates the editor view containing the static architecture of the drive model.
Then, the requirements are broken down to the components of the system and coded into assumptions and guarantees. Taking the phase current sensor as an example, typical requirements for these are (i) the delay of the (analog) output value w.r.t. the true value, (ii) its accuracy or imprecision w.r.t the true value, and their (iii) safety integrity level. These properties are typical examples of properties that are today still difficult to prove formally, but which can be judged by experts (e.g. using timed tolerance bands, evaluating step responses, etc.). For each signal, the architect proposes an assertion. For the component that produces the signal at its output, it becomes a requirement to guarantee the denoted signal properties, and the component that consumes the signal at its input may be designed under the assumption that the signal fulfills the denoted properties. Thereby, the architect creates a contract between producer and consumer component. How to distribute e.g. the allowed overall delay of a safety reaction onto the components is left to his free decision, based on his experience and on negotiation.

In our example, the producer component is the set of current sensors with its related analog signal conditioning (“PhaseCurrentSensors”), the signal is the measurement value at the input of the microcontroller and the consumer component is the microcontroller (“µCWithSW”). Listing 2 shows these refined requirements modeled as assumptions.

Assumptions for the µCWithSW on the PhaseCurrentSensor:
µCWithSW.phaseCurrentValues.delay <= 20 ms
µCWithSW.phaseCurrentValues.imprecision <= 2mA
µCWithSW.phaseCurrentValues.ASIL >= B

Listing 2: Assumptions for the phase current sensor

The guarantees of a candidate component for “PhaseCurrentSensors” can be stated by the system architect, e.g. based on the datasheets. They are provided in Listing 3.
Guarantees given by the PhaseCurrentSensors:

\[ \text{phaseCurrentSensors.phaseCurrentValues.delay} \leq 15 \text{ ms} \]
\[ \text{phaseCurrentSensors.phaseCurrentValues.imprecision} \leq 1 \text{ mA} \]
\[ \text{phaseCurrentSensors.phaseCurrentValues.ASIL} \geq B \]

Listing 3: Guarantees for the phase current sensor

Once the system is modeled and annotated with assumptions and guarantees, it is automatically analyzed by the analysis engine. Figure 4 illustrates the \textit{Results View} after the analysis engine was run. For the above example the result table lists three rows, one for each contract. In this case all contracts are evaluated to 'OK'.

The analysis traverses the signal connections derived from the static architecture and checks that all assumptions and guarantees are syntactically correct, that for each assumption a corresponding guarantee is found and that the assumption implies the guarantees. It is automatically evaluated, that the delay property guaranteed by PhaseCurrentSensors is sufficiently good for the needs of \(\mu\)CWithSW, because 15 ms is better than the required 20 ms, and a semantics for ‘better than’ regarding delays (in this case a lower value is better) has been implemented in the tool. In contrast, due to the lack of full formal semantics, it cannot be checked automatically if the sensor is actually able to deliver the signal within 15 ms and if the microcontroller is actually able to perform some safety reaction on time whenever it gets the current values within a delay of 20 ms. This assessment is still left to the expert’s manual evaluation, e.g. by calculations, simulations, or by integration testing.

For evaluating the usability of the method and the tool, we have presented the tool and the case study to several safety consultants of our company: they found the considerations on the electric drive valid, and after initial instructions, they were all able to specify some properties on their own and to interpret the check results from the tool.

6 Conclusion and Further Research Agenda

In this paper, we have proposed a procedure to apply Interface Contracts to the task of requirements decomposition and allocation onto components in the domain of safety-critical automotive systems. We provided an automated method to check the consistency
of the system architecture and we have developed a prototypical tool and applied all this to an electric vehicle drive as an example. The approach is promising, as we obtained positive feedback from several safety engineers who create safety concepts for automotive systems of numerous automotive OEMs and suppliers in their daily work. The ease of use is due to the graphical user interface with familiar component representation and the annotation language that is similar to well-known OO programming languages and allows the usage of physical units. In the future, the static architecture could be imported from existing UML/SysML design tools or Simulink, and signals could be imported from a data dictionary usually available.

We plan to extend the approach to further properties of port signals, but also to contracts that do not apply to ports, but to the component as a whole (component contracts). An example could be the assurance of sufficient processing resources for a software component.

Further it seems promising to combine this contribution with other recent development activities of ours: in a bachelor thesis, we have been performing experiments with parametrized templates for requirements, allowing direct references to signal names from a signal dictionary [Kü10]. Some of these templates appear to be promising candidates for requirements to the top-level systems that can be automatically translated into our annotation language. Secondly, we developed a tool that is able to extract the component hierarchy and the signal connections from Simulink tools and to create Component Fault Tree frames out of it [Ra14]. We plan to integrate these building bricks into a comprehensive Eclipse-based tool-chain for system and safety development, allowing the specification of contracts for the normal system function, the analysis of failures defined in terms of violations of these contracts, and the addition of safety contracts that describe the mitigation of the effects of these failures by additional safety mechanisms.

References


Acknowledgement: This research has been partly supported by the BMBF (German Federal Ministry of Education and Research) funded research project SPES_XT.