Model Maturity Levels for Embedded Systems Development, Or: Working with Warnings

Martin Große-Rhode

Fraunhofer Institute for Software and Systems Engineering, Berlin, Germany
martin.grosse-rhode@isst.fraunhofer.de

Abstract. The more modelling substitutes programming the more modelling tools should become development environments. Beyond enforcing the syntactic correctness of models tools should support a methodologically guided development in which milestones are indicated and warnings are generated to inform the user about issues that are to be solved to reach these milestones. In this paper we present an approach from the embedded systems domain that is materialized by the implementation of a prototypical model development environment. It indicates model maturity levels that correspond to an underlying development method and shows in the model maturity view which elements or parts of the model do not yet reach a level and why they do not reach it.

1 Introduction

Program development environments have led to a substantial increase of productivity in the construction of software. Completion suggestions based on the grammar of the programming language and the code produced so far, navigation in large amounts of code according to different kinds of relations, like place of declaration or place of usage, and, of course, the indication of errors and suggestions how to correct them reduce the time needed to produce compilable code drastically. Beyond the necessary conditions to produce code that can be compiled warnings are generated by the environment that indicate code quality according to different criteria. These warnings can be used to produce better code, or be ignored if they are considered not relevant.

Model development deserves the same kind of comprehensive support. Whether modelling is employed to replace programming as in pure generative approaches or to support programming by stating requirements, designs, and algorithms concisely, the development of models is an engineering task and bears its own complexity. Therefore it is not enough to be able to build a model. Construction support, navigation, indication of errors and methodological support are needed, too.

The modelling language and tool used as example in this paper have been designed for the automotive domain, in particular the development of AUTOSAR systems (see [AUT]). The AUTOSAR extension language *aXLang* (see [aXBench]) is a component description language that is used in the early process stages to represent functional requirements as function components, then to map these to software components, and to describe their distribution onto hardware components. The latter two are also defined in the *aXLang* and can be mapped to AUTOSAR descriptions.

2 The Development Process

The general pattern of the *aXLang* development process is shown in Figure 1. A task is represented by a model that is to be completed in the next step. In order to do so several solutions are worked out as far as necessary to be able to judge whether the solution satisfies the task, and to evaluate the solutions to decide for the best one. The solutions are also represented as models, and the selected best solution defines the task for the next development step – until the modelling part of the process has finished and code is produced.

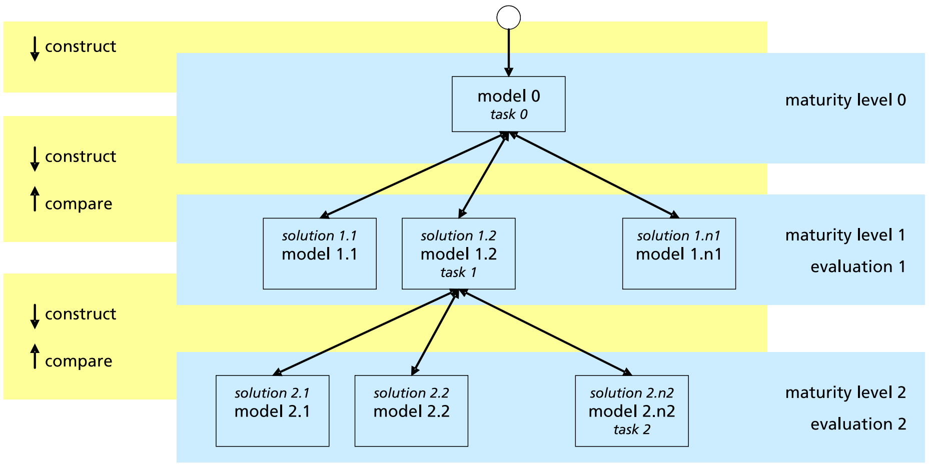

Fig. 1. Models as tasks and solutions in the development process.

There are two possibilities to decide whether a solution solves a task. Either an appropriate comparison operation on models is given that states whether a solution model solves a task model, or the development must make sure that the model is a solution by construction. In the *aXLang* approach the latter approach has been chosen, as discussed in Section 2.4.

The identification of the best solution requires appropriate evaluation operations. Furthermore, an indication is needed whether the models are both sufficiently and homogeneously detailed to yield comparable evaluation results. An estimation of the software size or the development effort for instance that is based on counting function points will only yield reliable results if functional designs are represented in the solution models at comparable levels. Otherwise the more detailed models always yield the worse estimations, independently of the adequacy of the design they represent. The indication of the appropriateness for an evaluation that demarcates a specific development stage is called a *model maturity level*. In general, development stages should always be defined by the evaluations that have to be passed and the validation operations that are possible at a stage.

In the *aXLang* process up to now the following model maturity levels are defined:

Level 1 Function Interface Model
Level 2 Function Simulation Model
Level 3 Deployment Model
Level 4 AUTOSAR Model

The first two are described in more detail in the following. They can be applied to any component description language. The third level is specific to languages that incorporate an application level and a resources layer. The fourth level is specific to AUTOSAR.

2.1 Function Interface Model

The function interface model is the first model constructed in an *aXLang* process. It represents one application function of an embedded system and specifies which information this function exchanges with the environment or other functions in the system. Its main usage is virtual integration, i. e. the check whether the application functions that make up the system according to their interfaces fit to each other. The model is derived from a use case analysis of the function. A function interface model has the following elements.

– one component, the one that represents the function;
– the input and output ports of the component;
– the logical signals and the operation calls the function shall accept or is allowed to deliver to other ones via its ports;
– the services of the component that represent the expected i/o-behaviours of the function;
– and its internal storages that are used to specify stateful functions.

As indicated in Figure 2 the elements of the model correspond to questions that should be posed to gather the functional requirements systematically. The model structure thus serves as a schema for the requirements elicitation.

At the function interface level the services are the use cases of the function. They are described informally by natural language texts, but constrained by a schema implemented in the language that guarantees that only the behaviour visible at the interface is described, and that only declared elements (ports, signals, operation calls, storages) are used for the description. The schema contains slots for the precondition, the interaction, and the postcondition of the service. Within the textual description references to ports, signals, operation calls, and storages

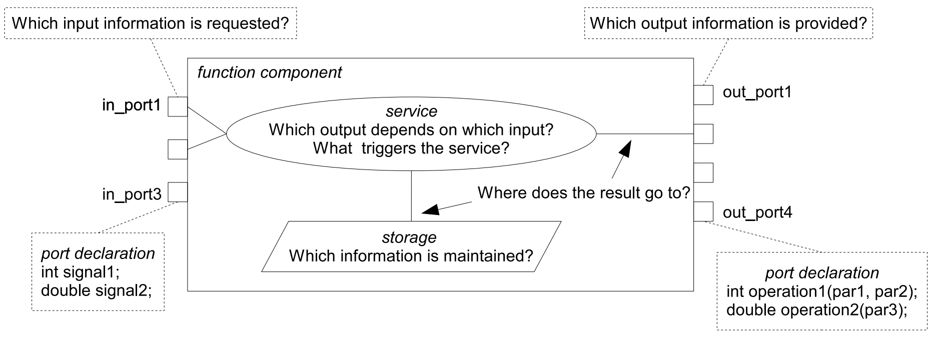
Which input information is requested?
Which output information is provided?
function component
in_port1
in_port3
out_port1
out_port4
service
Which output depends on which input?
What triggers the service?
Where does the result go to?
storage
Which information is maintained?
port declaration
int signal1;
double signal2;
port declaration
int operation1(par1, par2);
double operation2(par3);

Fig. 2. Function interface model as requirements elicitation schema.

are marked such that their declaration in the function interface model can be checked and renamings can be carried through as consistent refactorings.

As running example we use the function Condition Based Service (Cbs). It monitors the state of a vehicle and computes a summary of the overall state of the vehicle (green, yellow, red) and a car maintenance service date, i. e. a suggestion when to go next to the service. The function has been a case study in a project with the BMW Group (see [VEIA]). A graphical representation of its interface model is shown in Figure 3, the *aXLang* description in Table 1.

The behaviour description of the service *compute_cbs_data* is given as follows.

```
service compute_cbs_data  {
  ...
  behavior {
    precondition {$
      The ’ignition’ is on.
    $}

    interaction {$
      1. For each adaptive volume Cbs reads the ’relative_wear’
         from the corresponding sensor port.
      2. Cbs computes the ’service_date’ and the ’summary_estimation’.
    $}
  }
  ...
}
```

References are indicated by ’ ’, as in ’service_date’ and ’summary_estimation’.

Deriving function interfaces in this liberal but constrained way turned out to be very constructive in the industry projects in which a predecessor of the language has been used (see [Gro08]). The basic idea is that functions are understood best in terms of their behaviour and that the structure of the function can be elicited most concisely if it is based on a use case analysis. On the other hand, the method

```
top component Cbs {
  ports {
    in <ignition> pin_ignition;
    out <service_date, summary_estimation> pout_driver_interface;
    in <tick> pin_clock;
    in <cars_time, mileage> pin_board_data;
    in <relative_wear, initial_availability> pin_wheels;
    in <relative_wear, initial_availability> pin_motor_oil;
    optional in <initial_availability, relative_wear> pin_particle_filter;
    optional in <relative_wear, initial_availability> pin_spark_plug;
  }

  storages {
     storage cbs_data {
        int service_date;
        int summary_estimation;
      }
    }

  services {
    service display_service_date  {...}
    service compute_cbs_data  {...}
  }
}
```

Table 1. Interface model of the function CBS as *aXLang* text.

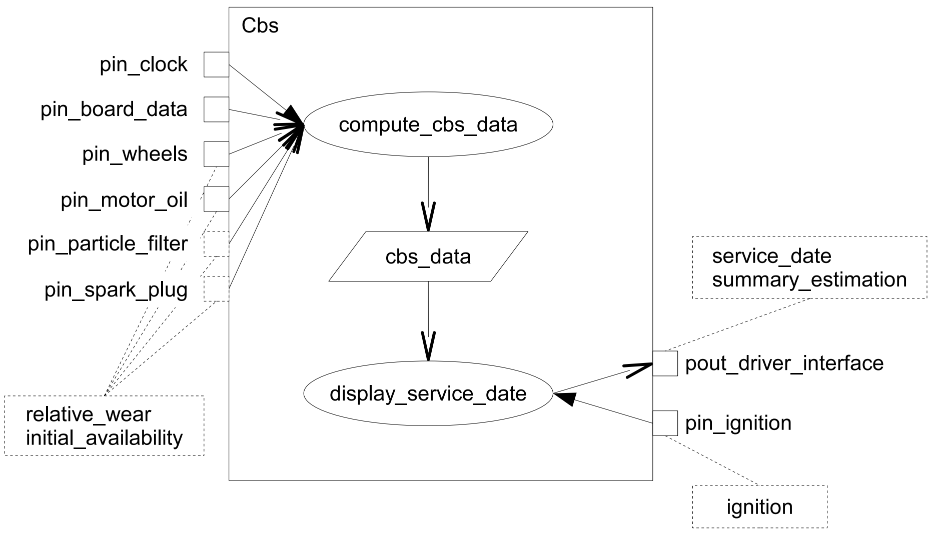

Fig. 3. Interface model of the function Cbs.

must make sure that behaviour descriptions are constrained to interactions with the environment; internal behavior must not be specified here. This is achieved in the *aXLang* by checking that each phrase in the behaviour description contains references to the elements declared in the interface model, i. e. each phrase must refer to an externally visible interaction. In the Cbs behaviour description the *ignition* signal is indicated as well as the input signals *relative_wear* and the output signals *service_date* and *summary_estimation*.

The usage of storages and the description of the access of a function to its storage in a use case might seem to contradict this principle. However, storages are considered as interface elements in the sense that they are only used as abstract means to describe that the function has a state. So the reader of a function interface model should be informed about the statefulness of the function, and in the refinement the storages must be refined and finally be implemented, too. In the Cbs example the storage of the Cbs data is used to decouple the continuous (periodic) computation of the Cbs data from its occasional display, triggered by the driver turning on the ignition.

The properties that are checked for the model maturity level *function interface model* are systematically derived from this methodological approach. Each element must be justified by its contribution to a use case. Since the model is a requirements model for the further development this strict rule itself is justified: Later on each element must be implemented, which results in development costs. Therefore no superfluous elements are allowed in the model.

The first set of properties that is checked is whether there is at least one service in the function, and whether each service has a use case (behaviour) description.

The second set of properties concerns the interconnection of the service with the structural elements of the function. Each service must have at least one trigger, which is given by a port and a signal or an operation call declared for that port. Moreover, the service must yield a result, i. e. there must be a port onto which the service writes an output or, in the case of a stateful function, there must be a storage to which the service delivers a result. Since these properties cannot be deduced automatically from the natural language descriptions the language contains service specification slots where read and write accesses to ports and storages are declared. The corresponding part of the specification of the Cbs service *compute_cbs_data* is:

```
service compute_cbs_data  {
  trigger pin_clock.tick;
  read pin_ignition.ignition;
  read pin_wheels.relative_wear;
  read pin_motor_oil.relative_wear;
  read pin_particle_filter.relative_wear;
  read pin_spark_plug.relative_wear;
  write service_date;
  write summary_estimation;
  behavior {...}
  ...
}
```

Checking the properties thus is a simple task; having these declarations in the model, however, is an important methodological contribution and within larger developments their indication in the maturity level view is indeed helpful.

Now, as mentioned above, it is checked whether all structural elements are justified by a use case. First for each port it is checked whether signal or operation calls are declared for this port at all; otherwise it is superfluous. Then it is checked whether the incoming signals and operation calls at the port are read by some service and whether the declared outgoing signals and operation calls are provided by some service. The analogous property is checked for the storages: each one must be both written and read by one or more services.

2.2 Dealing with Variants

Since the *aXLang* has been designed for the automotive domain it must provide means to deal with variants. At the architectural level, including the function interface models, variability can be expressed by alternatives, encapsulated in mutually exclusive elements (xor), optional elements, and parameterised elements (see [MR09]). In the case of a function interface model ports, signals, operation calls, and storages can be optional; services can be xor, i. e. product specific behaviours of a service can be specified.

Since the product specific behaviour and structure of a system in general cannot be localized to one place in the architecture but is spread over several components, feature models are employed to encapsulate the variance. An *aXLang* model altogether thus consists of several specific models. One is the *application model*, a model of the component architecture of the application view of the system. The function interface model is an application model at the first level of maturity; it represents an application function as one component. A second one is the *feature model* that characterizes the commonalities, differences, and dependencies of the different variants of the system in terms of abstract system features. The feature model is a tree of features indicating the mandatory, optional, and alternative features of the products of the system family. A mapping of the features to the application model defines which of the variant architecture elements are present in a system variant when a given configuration of features is selected. (For an introduction to feature oriented software product line engineering see [KLLK02].)

In the model of the Cbs function we have optional ports for the particle filter and spark plug sensor inputs because these are not present in all vehicles. They are indicated by the keyword *optional* (see Table 1). Whether one of the optional ports is present depends on whether the vehicle has a diesel or a gasoline engine. This is expressed in the feature model and the mapping of the feature model to the application model (*f2a_mapping*):

```
featuremodel CbsFeatures {
  features {
    xor engine {
      diesel;
      gasoline;
    }
  }
}
...
f2a_mapping CbsApplicationBinding CbsFeatures -> CbsApplication {
  \\ feature to port links
  f2p_links {
    engine.diesel -> pin_particle_filter;
    engine.gasoline -> pin_spark_plug;
  }
}
```

The feature mapping is estimated according to the same principle as above: each element must be justified. In this case this means first that each optional or alternative feature of the feature model must be mapped to an element of the function interface model and that each variant element of the function interface model must be bound by a feature. Furthermore the semantics of the features and the variant element must be respected: No mandatory feature must be mapped to a variant architecture element and no invariant architecture element must be bound by a variant feature.

2.3 Evaluation of Function Interface Models

As mentioned above a development stage should be defined by the evaluations that have to be performed and by the validation operations it allows.

The *validation operation* that becomes possible (and meaningful) with function interface models is virtual integration, i. e. the check whether the interfaces of the application functions of the systems fit to each other. For that purpose a system model is built by connecting the considered function interface models. More precisely: the ports of the function interface models are connected to specify which functions are senders and receivers of which signals and operation calls respectively. Communication with the environment is modelled by encapsulating the function interface models in a common super component (the system) and delegating the corresponding ports to the ports of the super component (see Figure 4). Composition and decomposition of components are discussed in more detail in section 2.4.

The necessary condition of the virtual integration is that each required signal and operation is provided, either within the system or by the environment. Input signals are required by a function, otherwise it would not be able to produce its output. Thus the connections in the system must be checked as to whether each signal is delivered somewhere and transported to the requesting function.

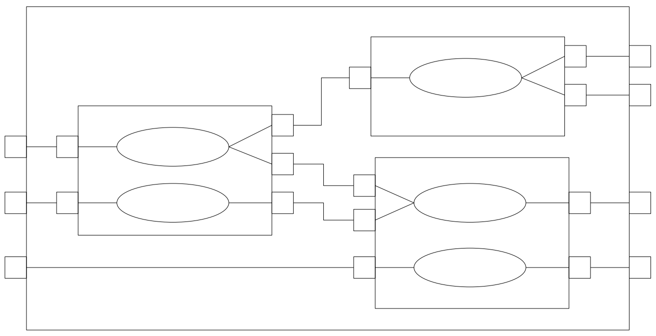

Fig. 4. Virtual integration of function interface models.

Operation calls are required by a function if they are sent from an output port. There must be a function that receives the function call at an input port, and operates it.

A sanity check can and should be performed here, too. If a function provides a signal at one of its output ports, there should be someone in the system or the environment who needs the signal; i. e. there must be a connection to the input port of a function where the signal is consumed, or an explicit delegation to the environment. Otherwise the specification would require the generation of a useless signal by the originating function – which produces development overhead. Since the implementor of the function typically does not receive the whole system model but only the model of the function she cannot check whether the required functionality is indeed needed. Analogously, operation calls at input ports of a function are checked: is there someone in the system or the environment who needs (calls) the operation? If no, remove it from the specification.

The *evaluations* of a function interface model implemented in the *aXBench*, the modelling environment for the *aXLang*, are estimations on the size of the software and the effort of its development. Both are based on a metric for system family models (see [KFS06]) that is an extension of the function point metrics to specifications including variance. The first estimation, the software size, is important for the cost estimation of the product (the necessary size of memories), the second one is important for the cost estimation of the process.

The *aXBench* furthermore provides an interface for the integration of other evaluation operations. Metrics that count elements as the one mentioned above, for instance, would get the elements of the model via the interface and deliver their results as a view to the *aXBench*.

2.4 Function Simulation Model

The behaviour of a function has been described in the function interface model in natural language only. The second milestone in its development is reached when an executable model is delivered.

In general an application function of an automotive system is too large as to be immediately modelled in such detail that the model can be executed. Therefore the model has to be decomposed into components representing parts of the function that are small enough to be provided with an executable description.

Decomposition is supported in the *aXLang* as in most other component or architecture description languages by component hierarchies. To allow the multiple use of subcomponents of the same type, the hierarchy is not directly represented in the language. Instead, components and subcomponents are different entities in the metamodel; a component (strongly) aggregates subcomponents and each subcomponent has a reference to a component that is its type. This encoding of hierarchies via instance-type relations is common in component or architecture description languages, as for instance in the UML composite structure diagrams, EAST-ADL, AADL, and AUTOSAR.

Executable behaviour is described in the *aXLang* by programming language code. Beyond the standard assignments and control structures it contains expressions for the access to the ports of the function. A write statement, used for the emission of signals and operation calls respectively, is of the form *write(port.signal, value)* or *write(port.operation, par_1_value, ..., par_n_value)*. An expression for reading a signal at a port has the form *read(port.signal)*. Operation parameters can be read in the function that received the call with *read(port.operation, par_j)*.

Checking the function simulation model maturity level first means to check whether each service of an atomic function has an executable behaviour description. Only atomic functions are checked because the decomposition overwrites the higher level description. The behaviour of the composed function is completely described by the composition of the behaviour of the subfunctions. The higher level function does not add behaviour to its parts, but just organizes their interconnection by connecting their interfaces.

The replacement of the function interface model by the function simulation model via decomposition implies the further checks that are performed to reach the function simulation model maturity level.

The first part is the structural decomposition. According to the definition of the language subcomponents can only be introduced and interconnected within the component that is decomposed. Thus the structural coincidence of the function interface model with the top level of the function simulation model is guaranteed by construction (see Figure 5).

What has to be checked, however, is whether the subcomponents are connected with each other correctly and whether they are connected with the higher level component correctly. Both amounts to checking the data flow in the composition, as in the virtual integration discussed above. Each required signal or operation call at the port of some subcomponent must be provided either by another subcomponent via a connection or by the super component via a delegation. To be economic, furthermore, each provided signal or operation must be requested by another subcomponent or the super component.

The second part of the check concerns the behavioural decomposition, or, to be more precise, the structural aspect of the behavioural decomposition. A

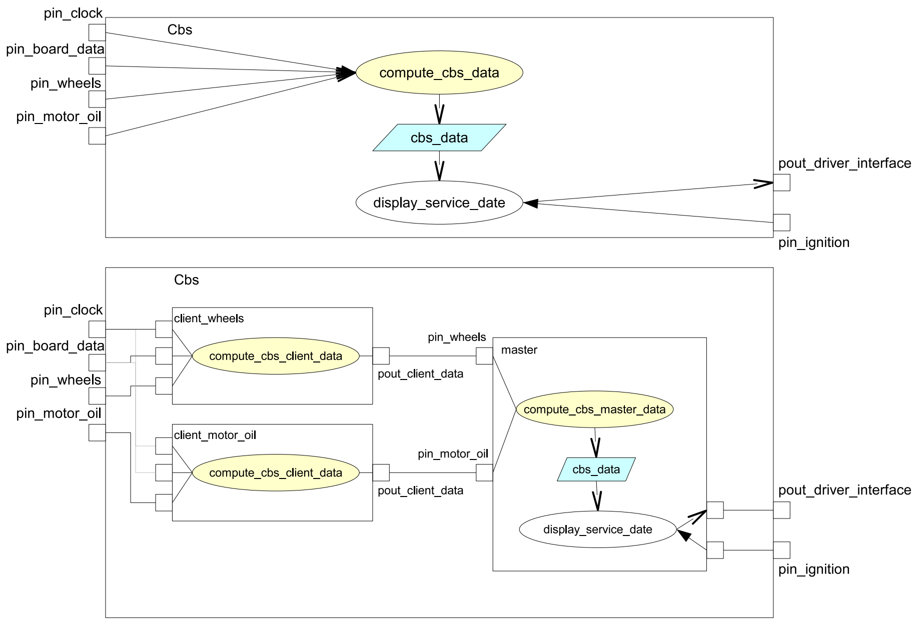

Fig. 5. Decomposition of services and storages.

service of the super component is refined by services of subcomponents, which means that the abstract (informal) specification of the super service is replaced by the more concrete (executable) specifications of the subservices that refine the super service. To state which subservices refine the super service, the language has a subservices slot for each service (see also Figure 5).

```
service compute_cbs_data  {
...
  subservices {
    clientWheels.compute_cbs_client_data;
    clientMotorOil.compute_cbs_client_data;
    clientParticleFilter.compute_cbs_client_data;
    clientSparkPlug.compute_cbs_client_data;
    master.compute_cbs_master_data;
  }
}
```

In the subservices slot only the *set* of refining subservices is given; the way in which they interact to realize the super service is determined by the way in which the containing subcomponents are connected. Thus there is no need to describe control structures in the subservices slot.

The first property that is checked for the maturity level is thus whether each service of the function interface model is decomposed, i. e. it has a non empty subservices slot. Next the declaration of the interconnection of the super service within the super component is checked: Are its triggers and read and write accesses correctly refined by the decomposition?

The decomposition must show the same effects at the function interface as the super service, i. e. it must neither introduce new inputs or outputs nor must it ignore inputs or outputs of the super service. If a decomposition of a service would require more input than the super service the integration of the implemented functions would fail. If it provides more output more implementation work than necessary would have to be done.

In order to check this property the data flow of the subcomponents according to the declaration of their services (read and write accesses *inside* the subcomponents) and their connections (data flow *in between* the subcomponents) has to be computed. With this information the read and write accesses of the composed subservices to the ports of the function interface can be compared with the read and write accesses of the super service declared in the function interface model.

Analogous to the decomposition of the services of the super component into services of the subcomponents the storages must be decomposed. For that purpose the *aXLang* provides a substorages slot in the specification of a storage:

```
top component Cbs {
  ...
  storages {
   storage cbs_data {
      int service_date;
      int summary_estimation;
      substorages {
        master.cbs_data {
          service_date -> master.cbs_data.service_date;
          summary_estimation -> master.cbs_data.summary_estimation;
        }
      }
    }
  }
}
```

In the example the storage is not distributed to subcomponents but resides in one single component, the master.

Using the substorages declaration also the correct decomposition of the read and write accesses of higher level services to their storages can be checked. If the super service has read or write access to a storage then at least one of its subservices must have an access of the same type to at least one of the substorages. Vice versa the subservices must not introduce more accesses than declared by the superservice.

Obviously the analysis of the decomposition is not complete. It covers only the structural declarations at the two levels. Whether the behaviour respects the declarations is another issue, which requires program or behaviour model analysis techniques that are not incorporated into the *aXBench* yet.

2.5 Refinement and Iteration

The check of the consistency of the function interface model and the function simulation model is based on the correctness of the decomposition. The basic idea thereby is to use hierarchical decomposition as refinement. Since the interface of the abstract model is fixed – additions are only made in the internal structure – we thus have substitutability by construction. In whatever way the function interface model is refined it fits structurally into the overall system. The decomposition information in the function interface model, i. e. the subservices and substorages slots, allows requirements tracing. They indicate the implementation (*composition of lower level services*) of a functional requirement (*description of a higher level service*) as well as the implementation of the required state properties.

In a development process, however, requirements typically are not entirely stable. One reason is that the more detailed design of a solution often reveals that, for instance, more input is needed by a function to compute its outputs, or that a restriction to less output would make the overall design more adequate. Having both the abstract super component (the requirements) and the subcomponents (the solution design) as hierarchy levels in one model supports the proliferation of requirements changes immediately. The maturity level check indicates whether new input signals for instance have been introduced at the sublevel but not yet delegated to the super level. Thus the user receives a warning that the interface of the super component has to be updated. Changing this interface must of course be reflected by a revision of the virtual integration, which can and should not be automated. However, the maturity level check provides the methodological support for the users' activities that yield concisely documented requirements change requests. The management of the changed requirements is best supported by organizational means in the process.

2.6 Simulating Models with Variants

The possible evaluations of the function simulation model are the same as the ones for the function interface model: counting elements to measure the predicted software size and development effort. The difference is that the basis for the estimation is now more detailed and thus the prediction more precise.

The major advantage of the function simulation model is that simulation becomes possible to validate and to debug the model. The *aXBench* has a simulation machine that uses the *aXLang* programming language description of the services' behaviours and the interconnection as expressed in the structure, i. e. the connection of the subcomponents.

A challenge in the automotive domain, as mentioned above, is dealing with variants. One possibility is to derive product specific models from the family model and then to simulate each of these. The *aXBench* has an operation that performs this derivation. Given an application model, a feature model, a feature-to-application mapping, and a feature configuration (i. e. a consistent subset of the feature model) it returns a new application model where all variant elements that are not bound to features in the configuration are removed. The removal of course respects all dependencies such that the result of the operation is a correct model again.

However, this procedure is tedious and neglects the advantages of product line engineering, namely to use only one model for all variants. A better solution is to provide a simulation that simulates all variants simultaneously, i. e. a simulation of the family model. The essential idea thereby is that each simulation run collects all configuration decisions that must be drawn in order to realize this run. Thus whenever a run encounters an xor component with its delegations to the alternatives it splits into all alternatives and memorizes in each branch that this alternative has been chosen. The result is then a tree of events where in each step the selected variant elements are indicated. This result can be used to identify behavioural invariants (commonalities) as well as to validate the variant specific behaviour (differences). The details of this system family model simulation are out of the scope of this paper, however.

3 Summary and Conclusion

Modelling is a part of the development process. In order to be useful it must be guided by a method and supported by a tool that does not only allow the construction of models but gives feedback on the state and the quality of the models.

The model maturity levels discussed in this paper are an effort to supply this kind of support, without constraining the development activity unduly. A distinction is made between syntactically correct models and models that – beyond that – represent milestones of the process. Error messages and correction suggestions are given in the case of violations of syntactic rules. Warnings are used to indicate what is missing in order to reach the maturity levels defined in the process. These warnings are grouped according to the checks that are preformed for the different levels, as discussed above. Within these groups the elements that are the causes for not passing a test are given and linked with the model editor such that corrections or amendments can be made immediately. Analogous to program development environments, the idea is to rise the efficiency of the modelling process by this support, and to achieve models of a better – since checked – quality.

Beyond the two maturity levels discussed in this paper two further ones are implemented in the *aXBench*. The first one, the deployment model maturity level, addresses models that contain a further specific model, the resource model. This one represents the computation and communication resources of the system, i. e. the nodes (electronic control units) and the buses and other communication means of the system. Similar to the feature-to-application mapping the *aXLang* supports the specification of application-to-resource mappings that define how the functions are allocated to the nodes and how the application level communication is realized by the communication infrastructure of the underlying system. The corresponding maturity level is checked according to the same principles as discussed above. Are all relevant model elements present; are all elements justified; are the semantics respected?

Having the resource level included in the model further evaluations are possible. A real time behaviour analysis for example can be made, provided information is given on the real time behaviour of the resources. A prototypical implementation of a schedule analysis algorithm has been used in the *aXBench* to illustrate the integration of an evaluation operation into the *aXBench* development process. The long term goal, however, is to use the *aXBench* interface to connect other, more professional evaluation tools.

The next maturity level indicates the AUTOSAR interface, i. e. the step in the process where the requirements and function design models of the *aXLang* can be handed over to the system generation process of the AUTOSAR methodology. The check of this maturity level is done constructively. The AUTOSAR export operation tries to translate an *aXLang* model to an AUTOSAR representation, and thereby collects all obstacles, i. e. all elements that cannot be translated to AUTOSAR. This yields the warnings of the AUTOSAR maturity level that are presented to the user in the maturity level view.

As discussed above the definition and implementation of a maturity level might not be technically challenging. Rather, a detailed analysis of the methodological role of the model's elements is required. The effect of the maturity level checks and the presentation of the results as a view in the tool, however, is considerable, as the programming development environments have shown.

References

[AUT] AUTOSAR development cooperation. AUTOSAR – Automotive Open System Architecture. www.autosar.org

[aXBench] aXBench-Homepage. The Autosar Extensible Workbench. axbench.isst.fraunhofer.de

[Gro08] Martin Große-Rhode. Methods for the Development of Architecture Models in the VEIA Reference Process. ISST-Bericht 85/08, Fraunhofer-Institut für Software- und Systemtechnik, May 2008.

[KFS06] Sebastian Kiebusch, Bogdan Franczyk, and Andreas Speck. An unadjusted size measurement of embedded software system families and its validation. *Software Process: Improvement and Practice*, 11(4):435–446, 2006.

[KLLK02] Kyo Chul Kang, K. Lee, J. Lee, and S. Kim. Feature oriented product line software engineering: Principles and guidelines. In *Domain Oriented Systems Development – Practices and Perspectives*. Gordon Breach Science Publishers, 2002.

[MR09] Stefan Mann and Georg Rock. Dealing with variability in architecture-descriptions to support automotive product lines. In David Benavides, Andreas Metzger, and Ulrich Eisenecker, editors, *Proc. 3rd Int. Workshop on Variability Modeling of Software-intensive Systems (VAMOS 2009)*, ICB-Research Report No. 29, pages 111–120.

[VEIA] VEIA-Homepage. Verteilte Entwicklung und Integration von Automotive-Produktlinien. veia.isst.fraunhofer.de