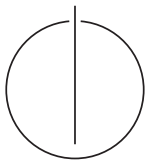
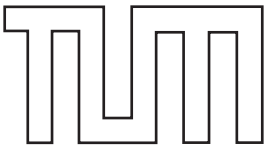


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



SPES 2020 Deliverable D1.2.B-5

On the Correlation between Functional and Logical Architecture

Author: Markus Herrmannsdörfer
Christian Leuxner
Sascha Schwind

Version: 1.0
Date: December 23, 2010
Status: Draft

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

Abstract—Within this paper, we address one of the pressing issues found in model-based software engineering, namely the consistency between different architectural views. We especially focus on the transition from the functional view—defining the functional requirements to a system—to the logical view—defining the logical implementation of a system. We outline and compare two possible solutions to support the consistency between those two views: the first one establishes links between the views after they have been populated, whereas the second one semi-automatically derives the logical view from the functional view.

Keywords—Abstraction Levels, Model Transformation, Refactoring, Integrated Development Process, Functional Architecture, Logical Architecture, Technical Architecture, Model-Driven Architecture, Adaptive Cruise Control

1 Introduction

Motivation. The idea of decomposing a system's specification along several 'levels of abstraction' constitutes good practice in the field of software engineering. This technique turned out to effectively reduce a specification's complexity and enables stakeholders to focus on certain aspects of the system, often called 'architectural views' or simply 'views', while abstracting from others. However, one major problem when managing different descriptions of the same system is a possible lack of consistency, i.e. the individual views of a specification—although being perfectly consistent in isolation—may impose contradicting requirements that can not be implemented. A means to avoid the introduction of such contradictions and to bridge the gap between different system views is to establish a precise relations (i) between the elements within a single view (intra-view) and (ii) between different views (inter-view).

Scope. In this paper, we focus on the inter-view consistency (ii). Moreover, we focus on the consistency between two dedicated views: the functional and logical view. The *functional* view defines the functions that a user can invoke at the boundary of the system. The *logical* view defines the internal implementation of the system as a hierarchy of components. Finally, we assume that the functional view is defined first, and the logical view is implemented based on the functional view. However, we believe that the approaches mentioned in this paper can also be applied to the consistency between other views.

Problem. Currently, it is largely unclear how to maintain the consistency between the functional and logical view. This inter-view consistency is important to ensure that the functions defined on the functional view are correctly implemented by the components on the logical view. However, ensuring the consistency between these views is a difficult endeavor, since the components may decompose a system in a completely different way than the functions. A function may be mapped to several components and a component may implement several functions. The missing link between functional and logical view prevents the seamless model-based development of systems.

Contribution. Within this paper, we present two promising approaches for ensuring the consistency between the functional and logical view: element links and model transformation.

The first approach, termed as *element links*, requires that the models of the functional and logical view are already existing. Usually, the relation of model elements across the different views only resides in the minds of the developer(s). This information can be made explicit by means of element links and hereby formally connecting various information of one common functionality of the system. This combined knowledge may be used to automate the static and dynamic analysis of the system. We suggest a methodology that facilitates the use of element links and explains which description technique should be applied in which view.

The second approach, termed as *model transformation*, provides a constructive approach to transition from the functional to the logical view. Starting with the functions defined at the functional view, we automatically derive an initial component architecture implementing the behavioral restrictions imposed by these functions at the logical view. However, this logical architecture is usually faraway from being implemented efficiently, e.g., since it may contain redundancies which must be factored out or it exhibits a structure that is not appropriate for the chosen target platform or the line-up of the development crew. In consequence, we propose to incrementally refactor this initial component architecture until the resulting specification satisfies the (non-functional) requirements imposed by the development process.

Outline. This paper is organized as follows. In Sec. 2, we introduce the notion of abstraction levels. In Sec. 3, we discuss an integration of the beforementioned abstraction levels, the related problems, and alternatives for their possible solutions. Finally, we conclude this paper with a discussion of promising directions in Sec. 4.

2 Architectural Views

A central question in software engineering is how to adequately structure the functionalities of the system at different levels of abstraction. A specification consists of a set of requirements, which usually deal with only fragmented aspects of the system behavior. Different scenarios and system modes (e.g., initialization, shutdown, or normal-case behavior) are usually described by separate requirements. Each further requirement adds a new aspect of the specified behavior. Specifications relate functionalities to each other, e.g., a scenario should always precede another one, or it has a higher priority than the other one. These relationships exclusively determine the structure of the functional specification. On the other hand, the functionality has to be realized by an architecture, which is typically structured into software components and hardware modules (e.g., electronic control units in a car). There may be many possible component-based architectures that implement the given functionality. Such architectures may be based on various architectural styles such as layered architecture or client-server. The pure functional structure has usually to be redistributed according to quality requirements and an existing hardware topology.

In light of this observation, a set of architectural views was introduced in [BKPS07, BH09, TRS+10], each view representing different kinds of information about the system (cf. Fig. 1).

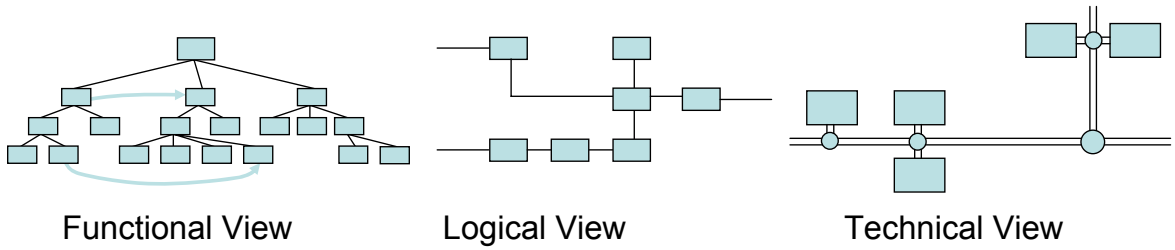


Figure 1: Architectural Views

According to [BH09], the functional view structures the user functionality into services without any architectural details. The specification consists of a set of services and relationships between them. A service specifies a partial and non-deterministic relation between certain inputs and outputs of the system, which interacts with its environment within a number of scenarios. In other words, a service is a fragmented aspect of the system behavior. The specification does not define the internal data flow within a system—every service obtains inputs from and sends outputs directly to the environment. Usually, services describe system reactions for only a certain subset of the inputs. This partial description allows the distribution of the system functionality over different services and/or leaving the reaction to certain inputs unspecified.

The logical view decomposes the functionality into a network of communicating components. Typically, the collaboration between the components realizes the black-box behavior specified at the functional level. This view forms the basis for grouping and distribution of the system functionality over components according to quality constraints rather than functional relationships. Here, structuring is done by means of the most diverse criteria. Besides functional decomposition the architecture might be structured according to the organizational structure within the company, or according to other non-functional requirements.

The technical view decomposes the system into a network of hardware modules (e.g., ECUs) and buses between them, on which the logical model is to be deployed. According to [TRS+10], in the technical perspective engineers need to consider hardware related issues such as throughput of communication, bandwidth, timing properties, the location of different hardware parts, or the exact interaction of the software system with the environment.

3 General Approaches

In this section, we propose two complementing approaches for relating the beforementioned *functional* and *logical views* in the context of reactive software systems. The first approach mainly dates from practical considerations in software engineering and requires more intervention from the designer. The second approach is quite ambitious in that it aims to relate both views in an automatic fashion, thereby following a correct-by-construction approach to model transformation. Each approach is shortly sketched within the following two sections, whereby all details of how to exactly realize one or the other is omitted. Indeed, in case of the automatic approach it is not totally obvious how a (sound) implementation should look like. Nevertheless, based on our experience in and many discussions about the topic, we believe that it is worthwhile to discuss some insights and open issues, and to conclude with a first

comparison balancing the fundamental pros and cons of both approaches at the end of this section.

3.1 Element Links

To address different viewpoints of a software system, various models are created during a development process. Typically, each viewpoint is concerned with a different aspect of the software system. For an instance, when a system is modeled from the functional viewpoint, the system is regarded as a black box, and only the system's interaction with its environment is taken into consideration. Usually, when a system is modeled by the use of different viewpoints, different description techniques are used as well. Frequently used description techniques are: use case diagrams, message sequence charts, component diagrams and statecharts. Each description technique has its advantages and limitations in regard to a certain viewpoint. It is important to understand which description technique supports which viewpoint, so that they can be applied properly.

The objective of this section is to briefly discuss the previously mentioned description techniques and to suggest at which viewpoint they may be useful. Furthermore we point out which elements — residing at different viewpoints and modeled by the use of various description techniques — correlate to each other. Due to the interdependencies of the elements residing in various heterogeneous models, we suggest to manually link the correlating elements, termed as *element links*, across different description techniques and thus across viewpoints. This consolidated knowledge may be used for analysis. Since the correlation between different elements is formally specified by the use of element links, the process of analysis can be automated.

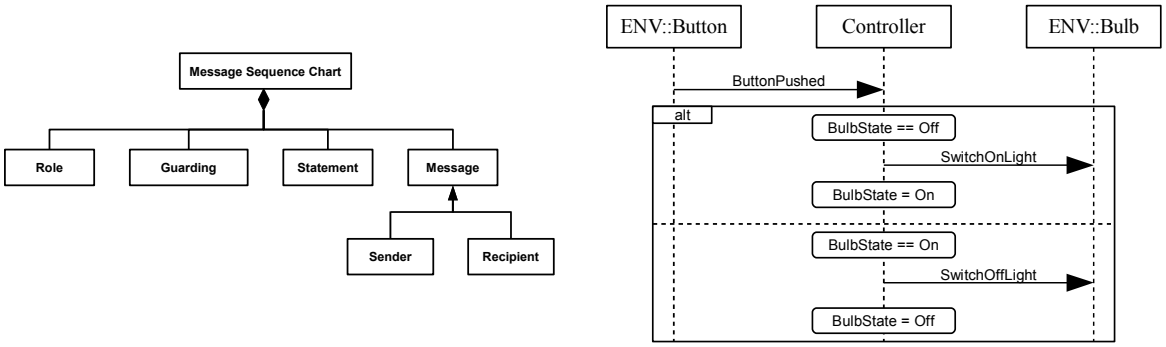
We do not consider our approach as a comprehensive methodology. It is regarded as a demonstration how a certain organization of description techniques may facilitate (automated) analysis be the means of element links.

In this paper we focus on a seamless model-based development process. We aim to generate fully executable source-code from models. The resulting source-code must not be further modified or completed by the developer(s).

3.1.1 Message Sequence Chart

Message sequence charts (MSCs) focus on the inter-component behavior of a software system. Specifically, on the interaction of software components with either other software components or hardware components. As mentioned earlier, the functional view is concerned with the description of the system's interaction with its environment. This interaction is usually specified within a number of scenarios. A scenario is a sequence of events of interests. MSCs are frequently used in practice to express scenarios [BL02, WS00]. Not only the intended behavior can be described, but also the forbidden behavior (referred to as *anti-scenarios*). This characteristics makes MSCs useful in describing the functionalities of a software systems. Thus they can be classified as a description technique that is adequate for modeling the functional view. Figure 2(a) gives an overview of essential model elements of a MSCs. The meta-model was simplified due to clarity and will be extended in future work. Figure 2(b) gives a concrete model of a MSC. It describe the following example application.

Example Application. To illustrate our approach we use a running example application throughout this paper. The example application describes the functionality of a light switch as following. If the button is pushed, then the light bulb is either switched on or switched off depending on its current status. I.e. if the light bulb is currently on [off], then it will be switched off [on] if the button is pushed.



(a) Meta-Model (b) Model

Figure 2: Message Sequence Chart

Since MSCs only describe partial behavior, additional models are required for describing the total behavior of a system [WS00]. Unless the total behavior is specified, fully executable source-code can not be generated. As stated earlier, we aim to generate fully executable source-code from design models that must not be manually modified.

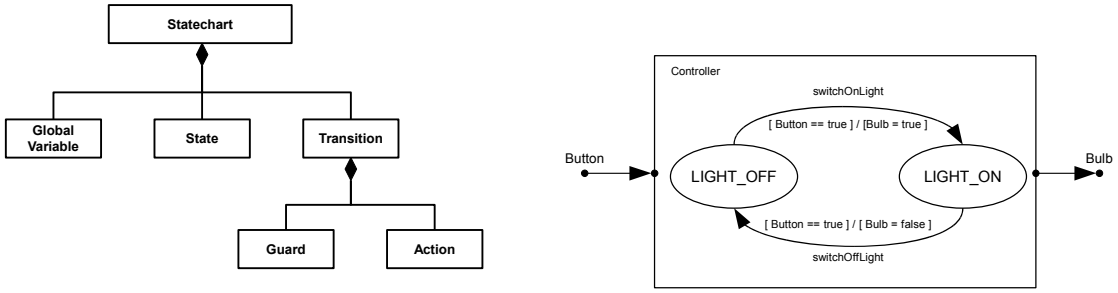
At this point we would like to mention, that we do not consider MSCs as the only mean for specifying the functional view. Other description techniques, like high level MSCs or activity diagrams may be useful as well. This will be investigated in future work.

A use case diagram may be used to structure the MSCs and to specify the dependencies between use cases. In this paper, use case diagrams are not discussed further and will also be positioned in future work.

3.1.2 Statechart

Statecharts are appropriated for modeling the behavior of a software system [HP98]. In contrary to MSCs, statecharts may be used to model the total behavior of software system and to generate fully executable source-code. MSCs and statecharts have overlapping concerns, but basically they are concerned with different aspects (respectively inter-component and intra-component behavior) of a software system. Attempts to (automatically) generate statecharts from MSCs have been made in research. But this approach is not discussed further in this paper. Since both description techniques specify the system from a different viewpoint they contain disjunct knowledge and must be modeled separately. Thus they can not be derived from each other. Hence they describe the system under different viewpoints, only combined they specify the system as a whole. Figure 3(a) gives an overview of essential model elements of a statechart. The meta-model was simplified due to clarity and will also be extended in future

work. In figure 3(b) a concrete statechart model is pictured that describes the intra-component behavior of the example application, that was describe earlier.



(a) Meta-Model (b) Model

Figure 3: Statechart

3.1.3 Component Diagrams

When there is enough information about the software architecture gained through the MSCs, the software architecture can be realized by using a component diagram.

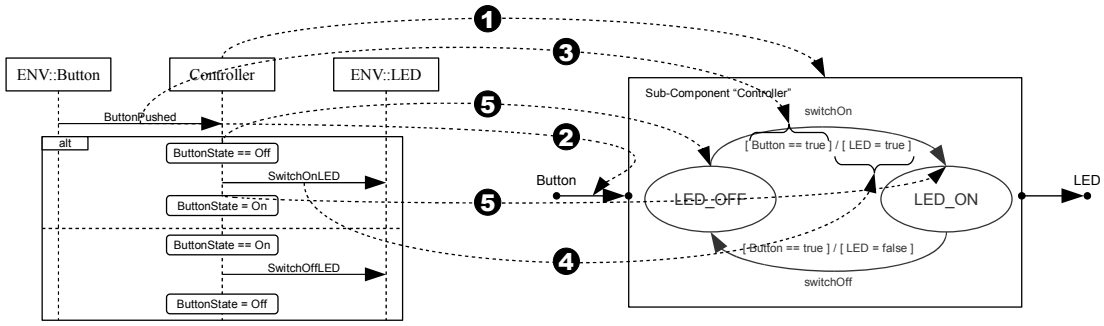
Component diagrams may be used to model the software architecture and to structure underlying statecharts. In figure 3(b) a component diagram is presented that contains a sub-component "Controller" which contains a statechart that defines the total behavior for this component.

3.1.4 Observed Correlation

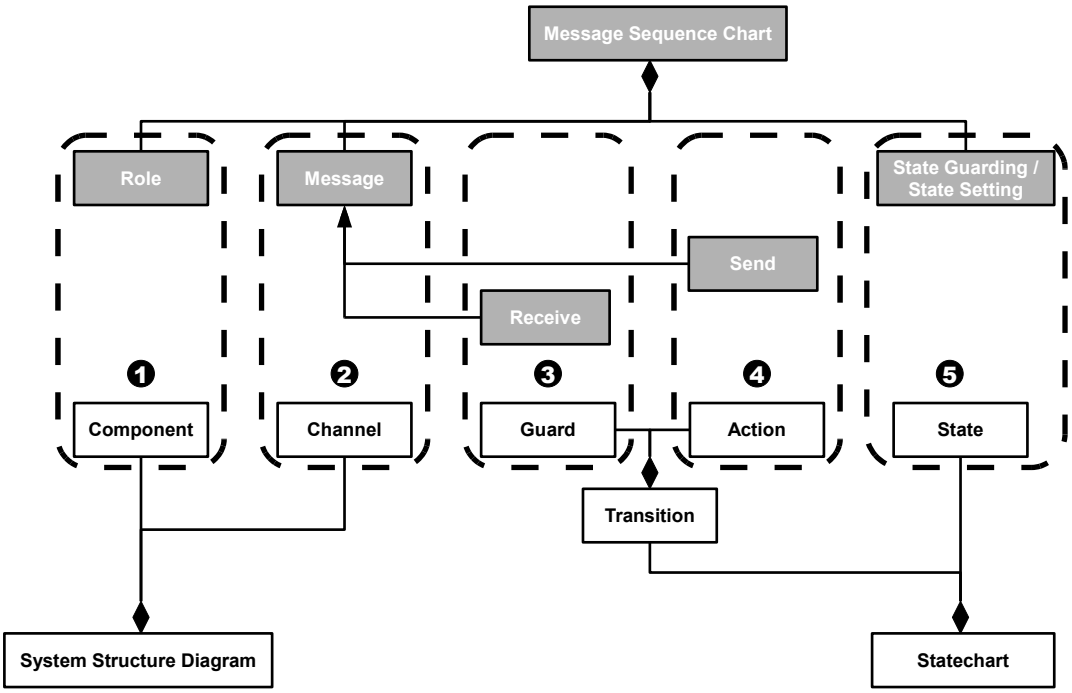
Table 1 and figure 4(a) give an overview of the observed correlation between the functional and the logical architecture based on the example application. The correlation is a kind of "realize" association. This is concretized in figure 4(b). The correlation may not always be unique. Under point 2 an 3, the Message <*ButtonPushed*> may be correlated either to channel <*Button*> or to guardian <*Button == true*>. In these cases the designer must decide which link is appropriate. Due to the small size and little complexity of the example application, the observed correlation will be evaluated in a proper case study in future work.

#	Model element in the functional architecture	Model element in the logical architecture
1	Role <*Controller*>	Component <*Controler*>
2	Message <*ButtonPushed*>	Channel <*Button*>
3	Message <*ButtonPushed*>	Guardian <*Button == true*>
4	Message <*SwitchOnLED*>	Action <*Bulb = true*>
5	State Setting <*ButtonState = On*>	State <*LED_ON*>

Table 1: Observed Correlation in the Example Application



(a) Model Correlation



(b) Meta-Model Correlation

Figure 4: Observed Correlation

3.2 Behavior-Conserving Model Transformation via Refactoring

A very appealing approach for relating different system views within the development process is to derive one from the other by means of a model transformation—deriving a logical from a functional system view is just one instance of this general case. But why is such a transformational approach so interesting? Because it requires only limited or even no intervention from designers and—in case of correct-by-construction transformations—enforces consistency between dependent specifications of a system. Thereby, a formal description of the system is translated schematically from a target into a destination model.

It is not surprising that, besides these obvious benefits, such an automated approach also comes with additional 'costs': both the source and the target model need to be formally founded w.r.t. syntax and semantics. Moreover, a precise correspondence between the source and target model needs to be defined, which states how the elements of the corresponding specifications relate to each other—both on the syntactic and the semantic level. To guarantee a sound approach, it must also be proven that the transformation does not change the model semantics. And there is the limitation that such a translation can of course only be fully automatic, if the source model already contains all necessary information to construct the destination. Consequently, a concrete specification can not be automatically derived from a specification which abstracts from necessary characteristics of the target (such as, e.g., distribution, data exchange, decomposition, etc.). In other words, only somehow 'similar' models might be transformed automatically.

Our first proposal for relating the functional and logical view comes in two parts: the first part (Sec. 3.2.1) comprises the rather straightforward construction of a 'canonical' component architecture from a functional system description. The exact details, in particular an algorithm implementing this construction, can be found in [Har10]. The second part of our proposed construction (Sec. 3.2.2) is more challenging: starting from an initial component architecture which was automatically constructed in the first step, we propose to incrementally apply a set of *refactorings* on the component level. Each such refactoring defines a binary relation $R(s,t)$ that transforms a source model s into an 'equivalent' target model t. We informally use the notion of *equivalence* to express that both models s and t expose exactly the same behavior ($s \approx t$). In other words, a refactoring does not change a model in the sense of adding or deleting any behavioral aspects: each possible behavior of the source model s is also a possible behavior of the target model t and vice versa. Thus, s and t only differ in their 'non-behavioral' aspects, such as, e.g., the distribution of functionality over components or their internal structure.

Our motivation for applying refactorings on the component level is quite simple: since—for technical reasons—the initial architecture resulting from the translation in the first step is far away from being implemented efficiently, we searched for a possibility to improve this initial architecture according to diverse quality criteria. During the refactoring process, an initial architecture may be 'tuned' incrementally until it finally converges to a behaviorally equivalent system architecture that better fulfills the quality criteria under consideration. The notion of composition on the component level exposes some pleasing algebraic properties that give the designer enough freedom to actually transform an architecture while still preserving its behavior.

Both steps of the translation are further detailed in the two subsequent sections.

3.2.1 Construction of a canonical component architecture

The starting point of the transformational approach is a formal specification of a system's functionality in terms of building blocks we call *functions*. A function formalizes the concept of a use case and describes some exemplary part of the system behavior which is observable at the system's interface (black-box view). Its description consists of two parts fixing the static and dynamic aspects of a function—called its *syntactic* and *semantic interface*. The syntactic interface constitutes the signature of a function and describes how a functionality of the system might be accessed from its environment, i.e. which data can be sent to and received from the system. The semantic interface defines the observable behavior of the function by defining the logical rules how data is processed and exchanged. In our setting, these rules are operationally specified in terms of I/O automata [HSE97]. To enable a modular description of a system's functionality, functions may be hierarchically decomposed into a set of sub-functions which execute simultaneously. This composition yields functional architectures—represented as trees as depicted in Figure 5.

✍ **Example 1** (`Adaptive Cruise Control`—Functional View)

Figure 5 illustrates the decomposition of a function that can nowadays be found in most premium vehicles: the Adaptive Cruise Control, `ACC`. The purpose of this function is to relieve the driver from controlling the vehicle's speed while still preserving a required minimal distance to other vehicles in front. The root function `ACC control` is thereby hierarchically decomposed into its sub-functions in a mode-oriented fashion, i.e. the functionality is decomposed into modes of operation which typically exclude each other: function `ACC control` is either in mode `on` or `off`; when the function is `on` and furthermore `activated`, the ACC operates either in `speed control` or in `distance control`, etc. Function `speed control` regulates the vehicle's speed to a value the driver has entered; `distance control` makes sure that a minimal distance to vehicles in front is not undercut. The reader may convince herself that the corresponding behaviors of these two functions are somehow 'conflicting', so both functions can not be activated simultaneously without causing contradicting effects (e.g., accelerate vehicle to required speed while car is in front).

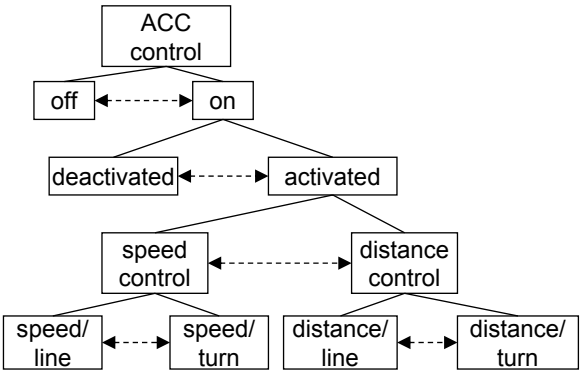


Figure 5: Starting Point of Transformation: Functional View of ACC [Har10]

□

As mentioned before, such a functional architecture can be canonically translated into an initial logical architecture. We do not provide the details of this translation, which can be found in [Har10]. Instead, we describe the outcome of the first translation step exemplarily in terms of the ACC.

✍ **Example 2** (`Adaptive Cruise Control`—Initial Logical View)

We consider the beforementioned sub-functions `speed control` and `distance control` of the `ACC control` functionality depicted in Figure 5. Applying the transformation algorithm of [Har10] to these two functions yields the first logical architecture depicted in Figure 6.

Therein, each function is injectively mapped onto a logical component with a coinciding syntactic interface[1]. Moreover, since `speed control` and `distance control` represent two operation modes of the ACC system which execute in a pairwise exclusive fashion (represented by the dashed arrow in Figure 5), additional auxiliary components such as `Priority` and `Mux` need to be introduced for technical reasons. The purpose of component `Priority` is to make sure that both functionalities are executed exclusively, i.e. either function `Speed Control` or `Distance Control` is executed, but not both. The purpose of component `Mux` is to merge the reactions of both functions. The necessity of merging a function's reactions stems from the fact that different functions may share outputs, whereas the output of a component is always uniquely determined.

Note that each function as well as each component additionally need to be specified by an I/O automaton. We deliberately abandon to present the corresponding automata in this report in order to keep things simple. Nevertheless, our transformational approach must deal with them and the algorithm in [Har10] indeed does so.

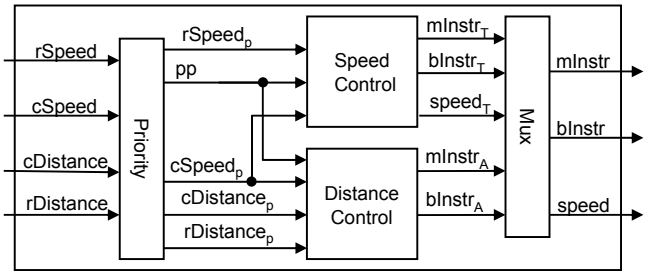


Figure 6: Extract from Initial Logical View of ACC [Har10]

□

The ACC example above already indicates that the architecture obtained from the previous translation does not constitute an 'optimal' design, since it may contain redundancies within the components and technical overhead as auxiliary components which among other things complicate the maintenance of the resulting system. We propose to tackle those complications by applying incremental refactorings to the initial architecture as described within the subsequent section.

[1] In Figure 5, the interfaces of functions are omitted to enhance the readability of functional architectures.

3.2.2 Refactoring component architectures

Once an initial component architecture is available, one might evaluate this architecture according to a variety of *quality criteria*. In the following, we list some common criteria according to Bass et al. [BCK03]:

Availability is concerned with system failure and its associated concequences. The availability of a system is the probability that it will be operational when it is needed.

Modifiability is about the cost of change.

Performance is about timing. Performance is concerned with how long it takes the system to respond when an event occurs.

Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users.

Testability refers to the ease with which a system can be made to demonstrate its faults through testing.

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides.

Refactorings are in-place transformations on the model that preserve certain properties. Typically, the behavior observable by the user as well as certain safety and liveness properties should be preserved. Refactorings are performed to improve the quality of the model w.r.t. to some quality aspect. We survey some typical model refactorings in the following and also provide a first categorization into *structural*, *behavioral*, and *other* refactorings.

Structural Refactorings. As their name already suggests, structural refactorings are concerned with the (internal) composition of models. Structural refactorings are not intended to change the timing and the behavior of the corresponding logical architecture. We distinguish between the following sub-categories:

Component Refactorings. The following refactorings act on the component hierarchy as defined by system structure diagrams (SSD), which is a representation of the logical architecture within the case tool AutoFocus [HSE97]:

- Push Down Component: A component is moved into a non-atomic component of the same SSD. As a consequence, the component is pushed down the hierarchy by one level.
- Pull Up Component: A component is moved up the hierarchy by one level. This refactoring is the inverse of Push Down Component.
- Wrap Components: A set of components within the same SSD is extracted into a new component.
- Move Component: A component is moved to an arbitrary position in the component hierarchy.
- Remove Single Component Hierarchy: A component which has only one sub component is inlined.

State Refactorings. The following refactorings act on the state hierarchy as defined by state transition diagrams (STD), a representation of I/O automata in AutoFocus:

- Push Down State: A state is moved into a non-atomic state of the same STD. As a consequence, the state is pushed down the hierarchy by one level.
- Pull Up State: A state is moved up the hierarchy by one level. This refactoring is the inverse of Push Down State.
- Wrap States: A set of states within the same STD is extracted into a new state.
- Move State: A state is moved to an arbitrary position in the state hierarchy.
- Remove Single State Hierarchy: A state which has only one sub state is inlined.

Behavioral Refactorings. Behavioral refactorings do change the time behavior of the AutoFocus model, but do not change the behavior if time is abstracted away. The following refactorings act on SSD and/or STD:

- Remove Intermediate State: An unnecessary intermediate state is removed.
- Insert Intermediate State: A new intermediate state is added which is necessary for the extension of the behavior.
- Split Component: An atomic component is split into two atomic components including their state hierarchies. In order to do so, an activation mechanism between the two components is introduced.
- Remove Inactive States: If possible, the activation mechanism introduced by Split Component is removed.

Other Refactorings. The following other refactorings can be thought of:

- Removing unused ports, channels or variables
- Removing non-reachable states or unfeasible transitions
- Minimalizing state machines
- Calculating product state machines

Having listed the most common refactorings that come into mind in the context of component architectures described by I/O automata, we now shortly examine how the application of these refactorings might influence the logical architecture in terms of our running example.

✍ **Example 3** (`Adaptive Cruise Control`—Refactored Logical View)

Starting from the initial architecture depicted in Figure 6, we could have decided that it exposes too much redundancy, thus complicating its maintenance: a closer look at components `speed control` and `distance control` reveals that they share a good portion of similar behavior that could be factored out. We might also decide to reduce the overall number of components within the architecture and therefore get rid of some of the auxiliary functions introduced by the transformation. To accomplish this task, we go through the list of the beforementioned refactorings and apply them to the corresponding parts of the model by some kind of pattern matching. For instance, since we want to reduce the redundancy within the components `speed control` and `distance control`, we apply the behavioral refactoring "Split Component" in order to extract some portion of the respective component behavior into a new `preprocessing` component. Afterwards, we may use the component refactoring "Wrap Component" to merge this new preprocessing functionality with the `priority` component. In the end, this process may yield an architecture that looks pretty much like the one depicted in Figure 7[2]. Therein, we decided to outsource a good portion of the preprocessing of component `speed control` and `distance control` into an individual component `preprocessing`. This corresponds to a pipes-and-filters design that divides the overall functionality sequentially into a common preprocessing and the differing postprocessings of the original components. By outsourcing common parts of the model into separate components, the *reusability* and *maintainability* of the resulting architecture tends to get better—so we optimized our architecture w.r.t. to some quality attributes of [BCK03] during the refactoring process.

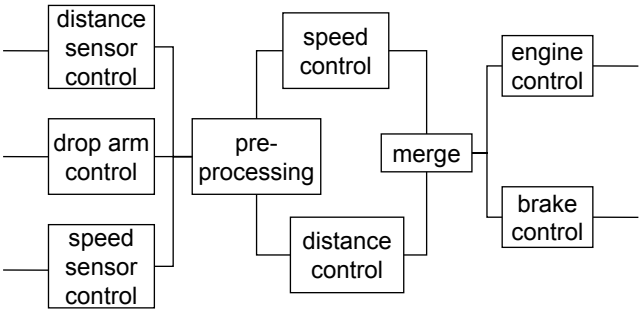


Figure 7: Refactored Logical View of ACC

□

3.2.3 Discussion

This little example should motivate the usefulness of applying refactorings on the component level in order to improve the quality of an architecture without compromising its functionality. Obviously, what remains to be clarified are the details of how to apply such refactorings in a sound and purposeful manner. So far, we cannot give a definite answer to this question. But we want to sensitize the reader of the potentials of such an approach—and of course the corresponding open issues that need to be solved. Hence, we conclude this section by posing

[2]Note that the automata of the refactored version of components `speed control` and `distance control` have changed due to the application of the "Split Component" refactoring. However, we introduce no automata within this report for sake of simplicity

some questions which we believe are relevant within this context and need to be answered to yield a promising solution to the integration problem within model-based design.

- *Convergence.* Obviously, given an initial architecture s and a set of refactorings R, there are many ways of applying these refactorings to s. But what are the refactorings that bring us 'closer' to the desired target architecture t, i.e. which refactorings let s ultimately converge to t?
- *Reachability.* Starting from an initial architecture s, is it at all possible to end up in the desired architecture t by a finite sequence of transformation steps that are all refactorings?
- *Completeness.* If the reachability requirement can not be guaranteed, can we anyhow get from s to an *equivalent* target architecture t if some steps on the derivation are no refactorings?
- *Tracing.* Is it possible to automatically update trace links between different views when applying refactorings to one view? If it is, this approach could automatically provide the information required for the first approach.

4 Conclusion

We conclude this paper by discussing some relevant characteristics as well as advantages and limitations of the two presented approaches.

- *Constructive vs. Analytical.* The transformational approach is constructive in the sense of yielding a logical architecture, and that process is—to a large extent—purely automatic. Functional properties of the functional view are preserved by construction and need not be checked again on the logical view.

  In this light, establishing element links comes with a much more analytical flavor: those links can be exploited for analyzing the system, e.g., by extracting drivers, oracles, and instances for integration and system test.

- *Flexibility vs. Automation.* In order to preserve all properties of the functional view along the development process, we insist on refactorings rather than refinements or arbitrary transformations within the transformational approach. In some cases, one may want to relax this restriction, e.g., when one is interested in reducing some of the potential non-determinism exposed by a specification. In such cases, a refinement is more appropriate than a refactoring, since it may resolve such occurrences of non-determinisms by eliminating some of the non-deterministic behavioral variants—while still preserving all properties. However, the decision on which variant to drop within a specification constitutes a design decision and can not be automated. Consequently, by taking refinements into consideration, one inevitably gives up some potential for automation. By allowing arbitrary transformations (neither refactorings nor refinements) on a specification, one additionally looses property preservation.

  One the other hand, the element link approach comes with no automation at all—each tracing link must be set manually—but provides the most flexibility. So depending on the

actual requirements one is interested in, one should carefully balance the consequences resulting from each approach.

- *Completeness vs. Partiality.* To benefit from the automation provided by the transformational approach, one has to construct an input-enabled, functional behavior of the system, i.e. the functional view must not contain underspecification. However, this somehow subverts the very notion of functional specifications, whose intention is to release the designer from taking irrelevant details into account—in particular the complete functionality of the overall system.

  By establishing feature links between a functional and logical view of a system, one can also start by relating elements of partial specifications. However, as a drawback, the logical view must be given a priori. Otherwise, no feature links can be established. The transformational approach requires no logical view for constructing its initial component architecture; this view is only needed to control the refactoring process.

References

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[BH09] Jewgenij Botaschanjan and Alexander Harhurin. Integrating Functional and Architectural Views of Reactive Systems. In *Proceedings of CBSE'09*, volume 5582 of *LNCS*. Springer, 2009.

[BKPS07] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, Feb. 2007.

[BL02] L. Briand and Y. Labiche. A uml-based approach to system testing. *Softw. Syst. Model (2002)*, 2002.

[Har10] Alexander Harhurin. *Von separaten Interaktionsmustern zu konsistenten Spezifikationen reaktiver Systeme*. PhD thesis, Technische Universität München, 2010.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1998.

[HSE97] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313*, pages 122 – 141. Springer, 1997.

[TRS+10] Judith Thyssen, Daniel Ratiu, Wolfgang Schwitzer, Alexander Harhurin, Martin Feilkas, and Eike Thaden. A system for seamless abstraction layers for model-based development of embedded software. In *Proceedings of Envision 2020 Workshop*, 2010.

[WS00] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 314–323, New York, NY, USA, 2000. ACM.