Understanding and Closing the Gap between Requirements on System and Subsystem Level

Sabine Teufl
fortiss GmbH
München, Germany
teufl@fortiss.org

Wolfgang Böhm
Technische Universität München
Garching b. München, Germany
boehmw@in.tum.de

Ralf Pinger
Siemens AG
Infrastructure and Cities Sector
Braunschweig, Germany
ralf.pinger@siemens.com

***Abstract*—In systems engineering, the increasing complexity of systems is handled by decomposing systems into subsystems. As part of the decomposition typically more abstract system requirements are refined to more detailed subsystem requirements. Refining system requirements to subsystem requirements includes the two steps *interface refinement* on the system boundaries, and a *decomposition* of system requirements to subsystem requirements. In order to apply formal analysis and verification techniques on the refinement of requirements, a formal refinement specification is necessary.**

In this paper we show the results of an exploratory industrial case study provided by Siemens, where we analyzed the refinement from system to subsystem requirements. We show that formal refinement specifications can become very complex, when interface refinement and requirement decompositions are performed in one step. In order to reduce complexity in the formal refinement specification, we introduce a formal restructuring approach for requirements. The main benefits of this restructuring approach are twofold. It enables reuse of requirements and knowledge preservation on the system level when the system architecture changes. Furthermore, quality assurance of the refinement on system level can now be performed independently from the system decomposition.

***Index Terms*—Requirements engineering; Modelling language; Systems engineering; Case study**

I. Introduction

In a collaborative project between fortiss GmbH (fortiss), Technische Universität München (TUM), and the Rail Automation business unit of Siemens AG (Siemens) [1] we conducted a case study where we applied a model-based approach for systems development to a real-world productive system developed by Siemens.

We pursued two main goals with the case study: First, we wanted to evaluate the methods and tools available on a system of realistic size and complexity in a non-academic environment. Second, we were interested to see how model based development techniques, like formal analysis and verification, can be used to analyze such a system and detect possible flaws and shortcomings.

As part of this study we applied a formal model-based approach called ”Model-based Integrated Requirements specification and Analysis” (MIRA) [2]. MIRA gives a precise notion of functional requirements and their refinement (adding more detail) by integrating a formal specification method [3] and a formal method for refinement specification [4].

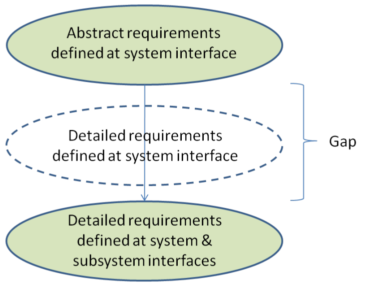
Abstract requirements defined at system interface
Detailed requirements defined at system interface
Gap
Detailed requirements defined at system & subsystem interfaces

Fig. 1. Detailed system requirements close the gap between system and subsystem requriements.

To reduce complexity, in systems engineering the system under development is decomposed into subsystems. Requirements are typically defined on system and subsystem level. Model-based requirements engineering promises to further reduce complexity by providing adequate models to structure and specify requirements. In order to apply techniques like formal analysis and verification, a formal specification of the refinement between system requirements and subsystem requirements has to be set up.

One of the main results of the case study was that the resulting formal refinement specifications between the given system and subsystem requirements became very complex, as in the documentation two distinct refinement operations (namely *interface refinement* from abstract to detailed interfaces, and *system decomposition*) from system to subsystem requirements) were performed within one refinement step. This complexity affects the specification and quality assurance of the refinement. We call the inclusion of both refinement operations in one refinement step a ”gap” between system and subsystem requirements.

In order to close the gap, we introduced a new level of requirements named *detailed system requirements*. These detailed system requirements connect system requirements with subsystem requirements. The detailed system requirements contain all interface refinement information of the subsystem requirements that are visible at system level. Fig. 1 depicts the refinement from system requirements to subsystem requirements. The new level of requirements not only simplifies the specification of the formal refinement specification, but also the quality assurance of the refinement. Furthermore, it enables reuse of the interface refinement on system level.

In this paper we report and discuss the results of our analysis. Background and related work is presented in Section II. In Section III we introduce the case study and the model-based requirements engineering approach. In Section IV we present the results of the case study, namely the analysis of the gap between system requirements and subsystem requirements, and discuss its implications on specification, quality assurance, and reuse. In Section V we introduce the restructuring approach to close the gap, again followed by a discussion of its implications. An example and the findings of the case study are presented in Section VI.

II. Background and Related Work

The FOCUS modeling theory [3] provides the background to formalize requirements and subsequently analyze and establish a formal refinement specification between refined requirements. The tool AutoFocus 3 [5], [6] is based on the semantics of FOCUS. AutoFocus 3 is a scientific open source tool for the component-based development of reactive, software-intensive, embedded systems. AutoFocus 3 provides a graphical representation of FOCUS.

The basic modeling element of AutoFocus 3 is a *component* as depicted in Fig. 2. A component has a syntactic interface described as a set of input and output *ports*. These ports connect components via *channels*. *Data types* are assigned to ports to restrict the messages allowed to be transmitted via the channels.

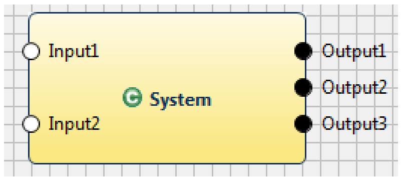

Fig. 2. Component specification

The behavior of a component can be described in two ways. Either, components can be described by a hierarchy of subcomponents, where the ports of subcomponents are connected with the main component and/or with each other. Alternatively, a component is defined by a behavior specification. The specification may be defined, e.g., in terms of specification patterns as introduced by Dwyer et al. [7] and as depicted in Fig. 3.

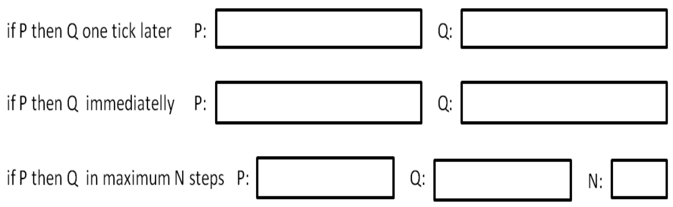
if P then Q one tick later P: Q:
if P then Q immediatelly P: Q:
if P then Q in maximum N steps P: Q: N:

Fig. 3. Specification pattern for formulas P and Q

In AutoFocus 3 the specification patterns are automatically translated to temporal logic formulas. The temporal logic formulas specify input/output traces of system behavior that need to be fulfilled by the implementation.

MIRA [2] is a framework for the model-based specification and analysis of requirements. It proposes models for representing and structuring the different RE work products. The models are described in Section III. MIRA is implemented as a plug-in for AutoFocus 3.

In FOCUS a refinement link between two formalized requirements can be represented formally as a *formal refinement specification* [4]. A formal refinement specification consists of a representation and an interpretation function. These functions define how the ports of an abstract requirement are mapped to the ports of a concrete requirement and vice versa (see Fig. 4). Based on these functions, the formal refinement specification defines the relations between the input and output ports of the formal representation of the requirements.

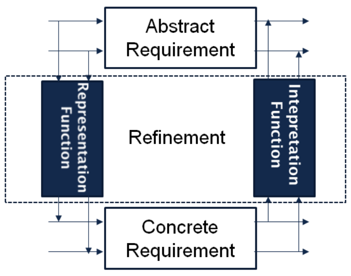

Fig. 4. A refinement specification specifies how the concrete requirement is related to the abstract requirement

Formal refinement specifications form the basis for formal analysis and application of verification methods. Setting up such a formal refinement specification can reveal syntactic flaws in requirements like inconsistencies, or incomplete coverage of abstract requirements by concrete requirements. The refinement relation can be verified by proving the logical implication between the interface assertions. In addition the specification can be used for the application of test cases on executable models [8].

Related work is available in the fields of system decomposition, formal refinement, and the analysis of reusable models.

DeSyRe [9] is an approach to systematically decompose system requirements into subsystem requirements according to a given system architecture with focus on informal or semi-formal requirements. It guides the requirements engineer through the decomposition of system requirements to subsystem requirements by proposing steps to manipulate the requirements. DeSyRE makes use of assumption / guarantee reasoning and decomposition patterns. We complement this approach by giving an a-posteriori analysis of suitable information contained in system and subsystem requirements that enables the reuse of the resulting requirements.

A well-known approach for the refinement of goals on systems level is KAOS [10]. Lamsweerde describes how requirements can be derived systematically from goals. He does not tackle the question about refinement from system to subsystem level.

Jazdi et al. [11] analyze the level of detail of models and model size in systems engineering, compared to the entire system, with respect to reuse for a domain engineering approach. In comparison to our approach they do not analyze the refinement of models but analyze the models independently. They recommend creating hierarchical, nested models which are coarse-grained at the top-level and fine-grained at the bottom-level. In comparison, we give detailed recommendations on the hierarchy of models for requirements.

III. Study

This section describes the case study performed in the project. We start describing the system under consideration and give information on our approach and the formal models used to specify the requirements and their refinements. We only focus on those parts of the approach that are relevant for this paper. A description of the overall project, its goals, set up, and methods used is given in [1].

Goal of the study was to apply a model-based approach for systems development to a system of realistic size and complexity. We were given the original (paper) specifications of the already existing Siemens system Trainguard MT (TGMT)[1] as input and were asked to re-implement a part of the system using pure model-based development methodology. We wanted to find out if the documentations produced for traditional development can serve as a basis in a pure model-based project as well and learn more about the limitations of the methods and tools. In order to keep the project at a size manageable by the available resources in the given time frame, we decided to focus only on a subsystem of TGMT.

We applied our scientific model-based requirements analysis approach (MIRA) to formally analyze the requirement specifications. In this paper we report on this part of the project and analyze our findings.

A. Study Object (see [1])

Trainguard MT is an automatic train control system for metros, rapid transit, commuter and light rail systems. It is a communication-based train control (CBTC) system with high-resolution train localization and bidirectional continuous data communication between the train and the wayside systems. By providing moving block train separation, optimal usage of the infrastructure is guaranteed. TGMT provides a large number of protection and automation functions for railway operation and uses components on the wayside and on-board the trains.

The implementation of the TGMT system concept is based on a cyclical exchange of position report telegrams sent from trains to the wayside subsystem and on movement authority telegrams sent from the wayside subsystem to the trains. Telegrams are standardized records that are digitally transmitted and usually used for remote control and for control purposes in system automation.

TGMT consists of two major subsystems: The wayside subsystem which calculates the movement authority on the basis of interlocking statuses and train position reports, and the on-board subsystem which supervises train operation within the given movement authority limit. The wayside and on-board subsystems use a track database, which stores track topography descriptions such as speed and gradient profiles. The on-board subsystem supervises and controls the train movement based on train localization, the information received from the wayside subsystem and the information stored in the track database.

[1]http://sie.ag/1aHfP1J

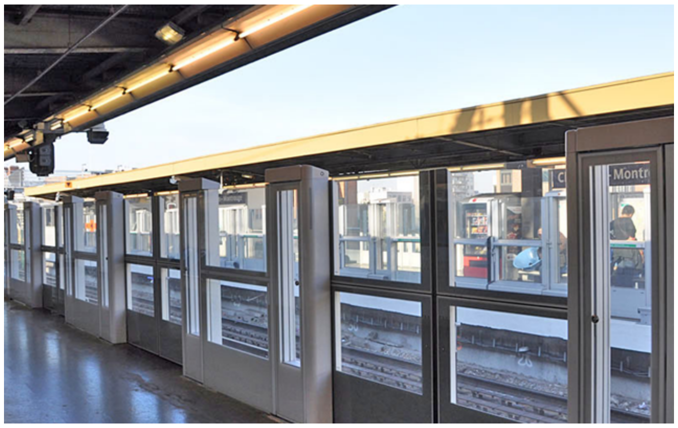

Fig. 5. Platform screen doors installed in Paris.

One purpose of TGMT is to control and protect passenger transfer at platforms. To this end, the system provides a function to operate platform screen doors (PSDs) for the protection of passengers in metro systems. PSDs are installed at the platform and can be implemented as full-height or half-height doors. Fig. 5 shows a typical half height platform screen door installation.

For the realization of PSD control opening and closing of the train doors and the wayside doors (the PSDs) has to be synchronized while taking care of passenger safety. To do this TGMT has an interface to the PSDs as well as to the train doors. For passenger safety the following protective mechanisms are part of the PSD function:

- The train doors as well as the PSDs are only allowed to open if the train is at standstill.
- The PSDs are only allowed to open when there is a train in the correct position at the platform (the train doors have to match the related PSDs).
- The train doors as well as the PSDs are only allowed to open on the correct side.
- Only those PSD sections are allowed to open that match the train length.
- During passenger transfer (open doors) the train must not move.
- If a PSD at a platform is unintentionally opened, no train is allowed to approach the platform.
- If there is a malfunction of the train doors, the PSDs must not open.

To model the PSD functionality, Siemens provided documentation which was taken directly and unchanged from their TGMT development. Among the documents was a *glossary*,

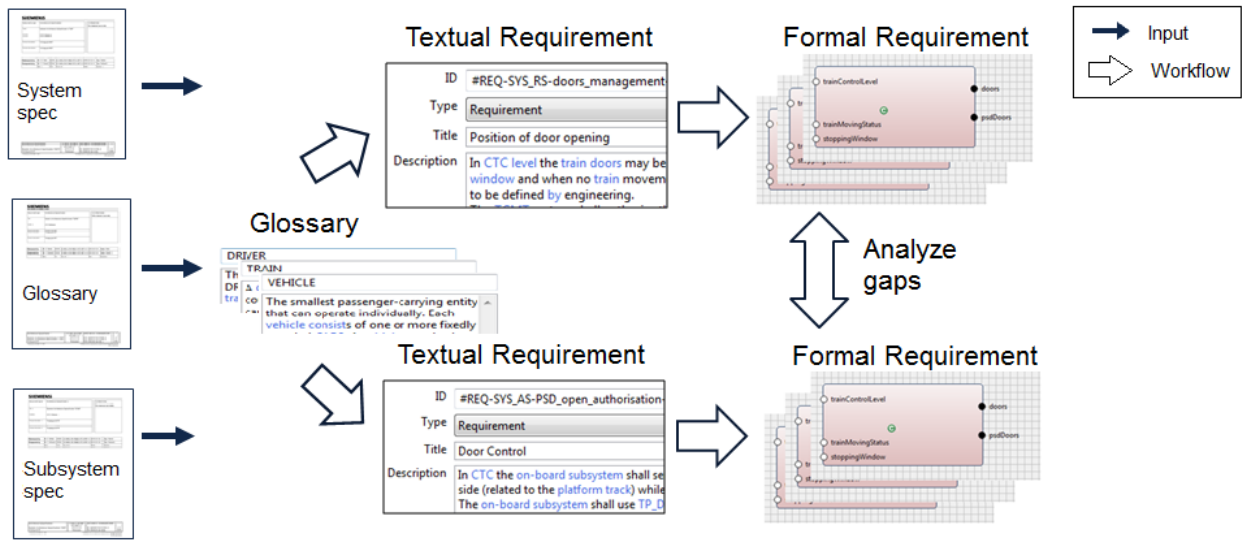
System spec
Glossary
Subsystem spec
Textual Requirement
ID #REQ-SYS_RS-doors_management
Type Requirement
Title Position of door opening
Description In CTC level the train doors may be window and when no train movem to be defined by engineering.
Glossary
DRIVER
TRAIN
VEHICLE
The smallest passenger-carrying entity that can operate individually. Each vehicle consists of one or more fixedly
Textual Requirement
ID #REQ-SYS_AS-PSD_open_authorisation
Type Requirement
Title Door Control
Description In CTC the on-board subsystem shall se side (related to the platform track) while The on-board subsystem shall use TP_D
Formal Requirement
Formal Requirement
Analyze gaps
Input
Workflow

Fig. 6. The refinement analysis was performed on the formalized requirements that are based on the original requirements provided by Siemens

describing the terms used in the application domain, a *requirements specification*, defining abstract requirements on the TGMT system level, and a *system architecture specification*, which contained architectural decisions and requirements on a detailed level. These documents were used as input for the modeling approach.

B. Study Execution

As tool support for the modeling approach we used the tool AUTOFOCUS 3 as introduced in Section II.

To create the formal models we conducted the following steps as proposed by the MIRA approach (see Fig. 6).

1) Description of Domain Concepts: All major concepts of the domain and the terms used for the PSD system were described in a *glossary*. To understand the PSD system, we analyzed each requirement and identified relevant domain and system terms, e.g., terms that define stimuli and reactions on the system or subsystem boundaries and the black-box system behavior. Whenever available, we included the term definitions from the input documents into the AUTOFOCUS 3 model.

2) Textual Requirements: All requirements from the requirements specification and system architetcure specification documents were transferred word-by-word into AUTOFOCUS 3.

3) Formalization of Requirements: For the formalization of the requirements we followed the FOCUS approach proposed by [12]. Here requirements are modeled as components with typed input and output ports and associated behavior. In the following, when we talk about formalized requirements we refer to the associated components (with typed ports and behavior, represented by a function mapping the input ports to the output ports). To create such a component in AUTOFOCUS 3 we performed two steps:

- To formalize the syntactic structure of a requirement in terms of a syntactic interface we extracted stimuli and reactions mentioned in the requirements and modeled them as ports of the AUTOFOCUS 3 component with an associated data-type (see Fig. 2).
- The formalization of the desired behavior of a requirement was modeled by adding an interface behavior specification to the syntactic interface. We specified the behavior of a requirement by AUTOFOCUS 3 specification patterns (see Fig. 3).

4) Refinement of Requirements: The subsystem requirements documented in the system architecture specification were linked to the requirements of the system requirements, indicating a notion of refinement. We modeled these relations in an initial step as *refinement traces* in AUTOFOCUS 3. *Refinement traces* are a structural connection between requirements. For these refinement traces we analyzed and specified the formal refinement specification between system and subsystem requirements based on the formalized requirements.

IV. ANALYSIS TO UNDERSTAND THE GAP BETWEEN REQUIREMENTS

We analyzed the requirements of the Siemens TGMT system introduced in Section III with respect to their refinement. The requirements of both documents were related to each other by means of tracing links, i.e., a requirement in the system architecture specification document is annotated with a reference to a requirement of the system requirements specification. The input documents from Siemens specified requirements on two levels of abstraction: Abstract system requirements describe the desired system capabilities and properties on a rather abstract level (specified in the system requirements specification), while more detailed requirements were defined on subsystem level (specified in the system architecture specification document).

For example, the system requirement *doors management* defines an abstract stopping window, where the train has to stop within. The subsystem requirement *doors release* details the stopping window to a stopping point for the train endpoints

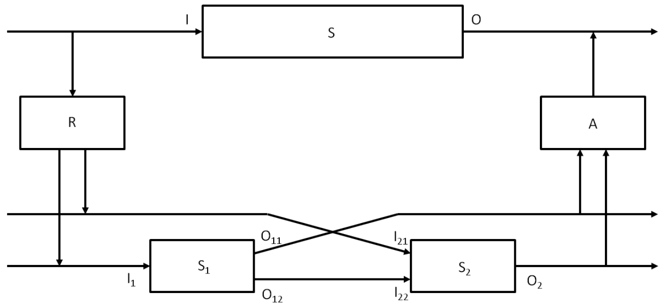

Fig. 7. A refinement specification specifies how the concrete requirements (S_1 and S_2) are related to the abstract requirement (S)

with plus and minus tolerances, while at the same time assigning the calculation of the stopping window to the on-board subsystem.

As discussed in Section III, we needed a formal refinement of the system level requirements in order to perform formal analysis and verification steps. For this purpose, we formalized the requirements from both documents by defining a syntactic interface and an associated interface behavior for each requirement of both documents, thus creating a formal representation of the specifications.

For the analysis assume two requirements formalized by their syntactic interfaces $(I \blacktriangleright O)$ and $(I' \blacktriangleright O')$, and behaviors S and S' respectively. As discussed above a formal refinement specification is defined as mappings of the input and output ports of the (formalized) requirements: a representation function R that maps the input ports and an interpretation function A that maps the output ports (see Fig. 4) in a way that the function $R \circ S' \circ A$ refines the behavior S (where the operator $\circ$ denotes a concatenation of the respective functions, see [4]). It is important to note that the theory just addresses relations between formalized requirements and does not address decomposition of systems into subsystems.

The situation complicates if (as in our case) the detailed requirements are specified on a subsystem level only. Fig. 7 gives an intuitive description of the situation.

A refinement of S could be given by the function $R \circ S_1 \circ S_2 \circ A$. Note that output O_{11} and input I_{21} are on the system interface, while the channel from output O_{12} to input I_{22} is within the system. On subsystem level there are two formalized requirements, requirement 1 with behavior S_1, input I_1, and output $O_1 = O_{11} \cup O_{12}$, requirement 2 with behavior S_2, input $I_2 = I_{21} \cup I_{22}$, and output O_2. We can derive that $R' \circ S_2 \circ A$ with $R' = R \circ S_1$ and $R \circ S_1 \circ A'$ with $A' = S_2 \circ A$ are refinements of S. This implies that the behavior S_1 of requirement 1 becomes part of the (formal) refinement function of requirement 2 (and similar the behavior S_2 for requirement 1).

From a practical point of view, in the architecture specification input document the requirement S is refined by two different operations that were taken in one step when moving from system requirements to subsystem requirements. We call these two steps *interface refinement* (a detailing of the system interface and its behavior) and *decomposition* of system requirements to subsystems (the break-down of requirements to subsystems without changing the system interface). In Fig. 7, we show a refinement of the behavior from S to $S_1 \circ S_2$, together with the refinement of input I to I_1 and I_{21}, and of output O to O_{11} and O_2. At the same time the system requirement is decomposed into S_1 and S_2.

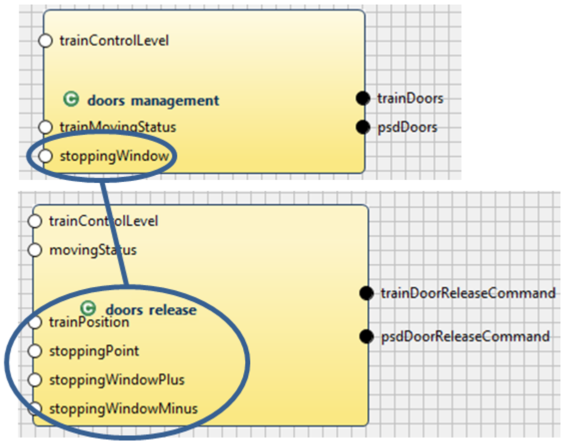

Fig. 8. Interface refinement on the system interface

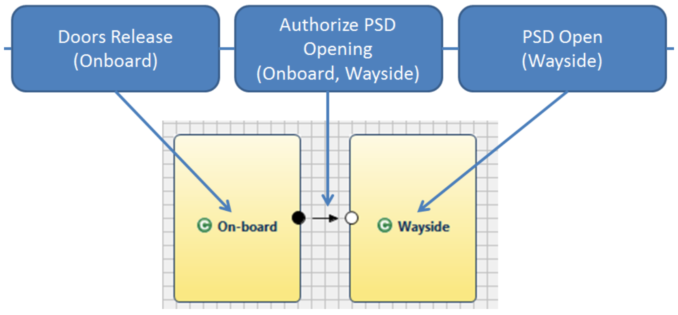

Fig. 9. Decomposition to subsystems

Fig. 8 gives an example for *interface refinement* from the case study. The inputs train position, stopping point, stopping window plus, and stopping window minus of the subsystem requirement *doors release* refine the input stopping window of the system requirement *doors management*.

As an example for *decomposition* refer to Fig. 9. The on-board requirement *doors release* defines that the conditions for opening the PSD door (e.g., the stopping window calculation) are calculated in the on-board subsystem. Requirement *authorize PSD opening* defines the communication between on-board and wayside subsystem. Requirement *PSD open* gives requirements on communication from the wayside subsystem to the external PSD system. Composing these three subsystem requirements gives the system behavior on the system interfaces. The detailed example is presented in Section VI.

We call the aggregation of these two different refinement operations a *gap* between requirements, because the interface refinement of a system requirement to a subsystem requirement cannot be defined without considering other subsystem requirements. A *gap* exists from the viewpoint of a stakeholder, e.g., a requirements engineer. In order to verify, whether the system behavior defined in the subsystem requirements (e.g.,

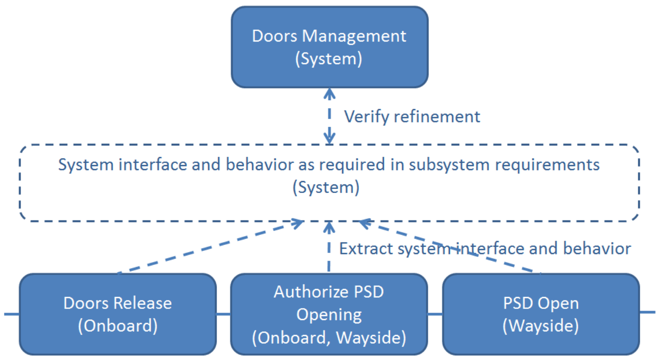
Doors Management (System)
Verify refinement
System interface and behavior as required in subsystem requirements (System)
Extract system interface and behavior
Doors Release (Onboard)
Authorize PSD Opening (Onboard, Wayside)
PSD Open (Wayside)

Fig. 10. Gap between system and subsystem requirements

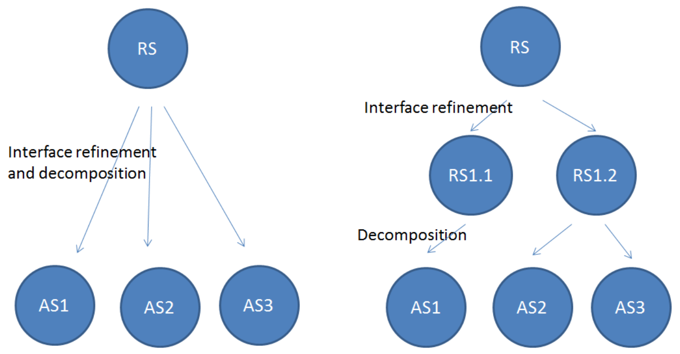

Fig. 11. Originally (left side), two levels of requirements were specified. We introduced a third layer of *detailed system requirements* (right side).

in *doors release*, *Authorize PSD opening*, and *PSD Open*) is a valid refinement of the system requirement, the engineer has to identify and analyze all subsystem requirements with internal communication in order to extract the information necessary for the verification as depicted in Fig. 10.

There are several consequences of such a gap:

Specification: In case of a gap defining the formal refinement specification is a challenging task: The information of all other subsystem requirements belonging to one decomposition has to be included in the formal refinement specification and to be integrated with the interface refinement (e.g., $R' = R \circ S_1$). This results in complex representation and interpretation functions and is a possible source of errors.

Quality assurance: If there is no formal refinement specification, as a consequence formal analysis and verification, as sketched in Section II, cannot be performed automatically. Decomposing requirements may lead to a scattering of system requirements (i.e. system behavior) over several subsystem requirements. In this case, analyzing requirements regarding their interface refinement is more challenging, as it is necessary to identify and consider all relevant subsystem requirements for the analysis.

For example, the validation of the interface refinement against the expectations of stakeholders is complex, as the stakeholders also have to incorporate the decomposition. Also manual checks on whether the behavior of the refined subsystem requirements conform to the behavior required in the system requirement are challenging as the decomposition has to be considered.

Reuse: Documenting system interface refinement only on subsystem level also hinders reuse of the refinement information when the architecture changes. In contrast, if the interface refinement is already documented on the system level, then reuse of concretization information is possible even when the decomposition of the system into subsystems changes.

V. Constructive Approach to Close the Gap between Requirements

In this section we introduce a bottom-up restructuring approach to close the gap between requirements on system and subsystem level. We propose to introduce a new level of requirements called *detailed system requirements*. Detailed system requirements have the same scope as the system requirements, i.e., they take a black-box view onto the system, while containing all relevant interface refinement information of the subsystem requirements. Hereby we separate the steps interface refinement and decomposition, thus eliminating the gap between the different levels of requirements.

Detailed system requirements define a connecting level between system and subsystem requirements. By taking only a black-box view on the system, detailed system requirements do not contain any details of the communication between the subsystems. In a second step, detailed system requirements are then decomposed to subsystem requirements. Fig. 11 visualizes input and result of the restructuring approach. Input of the case study were two levels of requirements (left side). The right side shows the results of the restructuring approach. RS denotes an abstract system requirement from the system requirements specification and AS denotes the detailed subsystem requirements from the architectural specification. RS1.1 and RS1.2 are the newly created detailed system requirements.

As sketched in Fig. 12 the abstract system requirement with behavior S and interfaces I and O is refined to the detailed system requirement with behavior S' and interfaces $I' = I_1 \cup I_{21}$, and $O' = O_{11} \cup O_2$. A refinement of S would be given by the function $R \circ S' \circ A$. The detailed system requirement is decomposed to the subsystem requirements, requirement 1 with behavior S_1, input I_1, and output $O1 = O_{11} \cup O_{12}$, requirement 2 with behavior S_2, input $I_2 = I_{21} \cup I_{22}$, and output O_2. A decomposition of S' would be given by the function $S_1 \circ S_2$. R_{id} and A_{id} denote the identity of the interfaces S' and $S_1 \circ S_2$.

The refactoring approach uses two operations on the subsystem requirements in order to obtain detailed system requirements:

- *Generalization:* The *generalization* operation changes the scope of a requirement from subsystem to system. However, the scope can only be changed, when all inputs and outputs of the requirement are on the system border.
- *Merging:* In order to perform the *generalization* operation, it may be necessary to *merge* requirements defining subsystem communication, so that the resulting new requirement relates inputs with outputs on system level only.

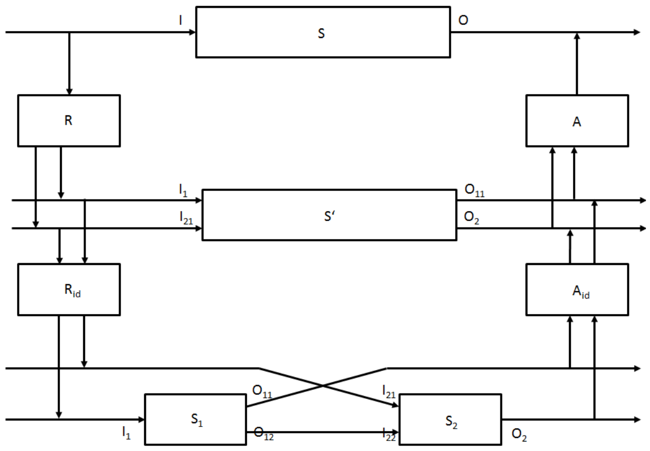

Fig. 12. The abstract requirement S has an interface refinement to S'. S' is decomposed to S_1 and S_2.

The refactoring approach starts bottom up from the detailed subsystem requirements and inverts the decomposition. In a first step all subsystem requirements are *merged* that are have common input or output ports within the system The merged requirement still contains information about subsystem communication. By *generalizing* the requirement, the subsystem communication is removed and the scope of the requirement is changed to the system. The result is the detailed system requirement.

The introduction of the detailed system requirement level requires additional specification effort as more requirements and more formal refinement specifications have to be specified. It increases the overall number of requirements and links between them. On the other hand, in Section IV we mentioned consequences related to those gaps. We now discuss how detailed system requirements can overcome the problems, which from our point of view justifies the additional effort:

Specification: By separating interface refinement and decomposition, information on subsystems is not integrated with interface refinement when specifying representation and interpretation functions. Therefore, these functions become easier to specify. This eliminates a possible source of specification errors.

Quality assurance: Quality assurance of the interface refinement, such as the validation against the expectations of the stakeholders, or the verification of requirements against the implementation, can now be performed on system level independently from the system decomposition.

Reuse: Detailed knowledge of the system interface is now available not only at subsystem level, but also on system level. This shift of knowledge from subsystem to system level enables reuse of the interface refinement information when changing the decomposition of the system into subsystems. The new requirements layer facilitates the understanding of the rationale behind decisions by separating interface refinement and decomposition. An additional benefit is the possibility to introduce exception/fault scenarios on system level based on the detailed system requirements.

VI. Case Study Example and Findings

In this section, we demonstrate our approach using an example from the Siemens case study. We apply the model-based analysis approach introduced in Section III. Then we present the results of the refinement analysis as discussed in Section IV. We apply the restructuring approach presented in Section V. Finally, we present the findings from our case study.

A. Introduction to the Example Requirements

In the Siemens documentation the requirement *door management* was detailed by 11 subsystem requirements. From this set of requirements, 6 requirements were related to the on-board subsystem, 4 requirements detailed the wayside subsystem, and one requirement defined the communication between both subsystems. From these subsystem requirements we selected the requirement *door release* as an example for a subsystem requirement.

The system requirement *door management* basically states that door-opening has to be authorized before the doors can actually be opened and gives condition for the authorization. Both, door authorization and door opening commands are calculated in the on-board subsystem. While the train doors are controlled by the on-board subsystem, the wayside subsystem controls the PSDs based on the train door commands. To enable this, the train door information is forwarded from the on-board to the wayside subsystem.

Note that the requirement *door management* corresponds to behavior S with syntactic interface $(I \blacktriangleright O)$ in Section IV. Requirement *door release* corresponds to behavior S_1 with syntactic interface $(I_1 \blacktriangleright O_1)$. The resulting detailed system requirement *detailed door management* corresponds to behavior S' with syntactic interface $(I' \blacktriangleright O')$

B. Application of MIRA

System: The system requirement *door management* states the conditions for the authorization of the doors.

> ”In train control level C the train and the PSD doors may be opened only if the train is located completely within a platform area within the predefined stopping window and when no train movement is detected. The train position allowing the door opening and the subset of doors allowed to open have to be defined by engineering.”

First, we identified terms describing the scope of the requirement, and terms describing the system boundaries for the *glossary*. Scope of the requirement is the *TGMT system*. Terms describing the system boundaries are for example *train control level*, and *stopping window*. In simplified terms, the train control level expresses the possible operating relationships between the wayside and on-board subsystems, while the stopping window is a supervised area on the platform track where door opening is allowed.

We formalized the requirement *door management* creating a requirement component which defines the syntactic interface as shown in Fig. 13, with input ports *trainControlLevel*, *trainMovingStatus*, and *stopWindow*, and output ports *trainDoors*,

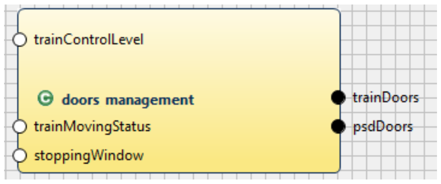

Fig. 13. Syntactic interface of system requirement *door management*

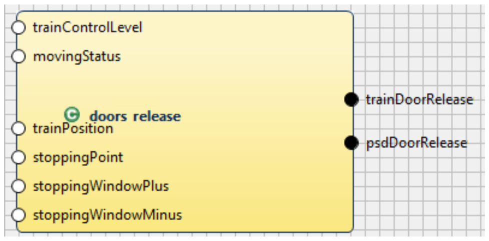

Fig. 14. Syntactic interface of subsystem requirement *doors release*

and *psdDoors*. The data types and their values are described in Table I.

The behavior (S) was formulated according the *IF...THEN* pattern. The formulation *only if* implies rules for closed doors.

$$\begin{aligned}&\text{If } \mathit{trainControlLevel} == \mathit{DT\_TrainControlLevel\_C(\ )}\ \&\&\\&\quad(\mathit{stopWindow} \neq \mathit{DT\_StopWindow\_Within(\ )}\ \|\\&\quad\mathit{trainMovingStatus} \neq \mathit{DT\_TrainMovingStatus\_Standstill(\ )})\\&\text{then } \mathit{trainDoors} == \mathit{DT\_Door\_Closed(\ )}\\&\text{after at most 1 ticks.}\end{aligned}$$

Note that here it is assumed that the system needs at most one processing step (tick) to react to the event.

Subsystem: We now show the subsystem requirement named *doors release*:

> ”In train control level C the on-board subsystem shall authorize the doors for opening (PSD as well as train doors) if the train has come to a safe standstill and the estimated train extremity is located within: platform stopping point minus stopping_window_minus and platform stopping point plus stopping_window_plus.”

Scope of this requirement is the *on-board subsystem*. In order to formalize this requirement we created a requirement component with input ports *trainControlLevel*, *trainMovingStatus*, *trainPos*, *stopPoint*, *stopWindowMinus*, and *stopWindowPlus*, and output ports *trainDoorRelease*, and *psdRelease*. The data types of the variables and their values are described in Table II.

The syntactic interface of the requirement *doors release* is depicted in Fig. 14.

C. Analysis of the Refinement

Based on the links between system and subsystem requirement given in the input documents we wanted to establish a formal refinement specification between the system requirement and the subsystem requirement. For that purpose we analyzed the ports and the associated data types.

Inputs *trainControlLevel* and input *trainMovingStatus* are identical on system and subsystem level. The input *stopWindow* of system requirement *doors management* is refined to the inputs *trainPos*, *stopPoint*, *stopWindowMinus*, and *stopWindowPlus* of subsystem *doors release*. All subsystem inputs of requirement requirement *doors release* are at the system interface (corresponding to input I_1 in Fig. 7). Denoting the inputs of the system requirement with I and of the subsystem requirement with I_1, the representation function is:

$$\begin{aligned}&I.\mathit{trainControlLevel} = I_1.\mathit{trainControlLevel};\\&I.\mathit{trainMovingStatus} = I_1.\mathit{trainMovingStatus};\\&\text{If } (I_1.\mathit{stopPoint} - I_1.\mathit{stopWindowMinus} < I_1.\mathit{trainPos}\ \&\&\\&\quad I_1.\mathit{trainPos} < I_1.\mathit{stopPoint} + I_1.\mathit{stopWindowPlus})\{\\&\quad I.\mathit{stopWindow} = \mathit{DT\_stopWindow\_Within};\}\\&\text{else}\{\\&\quad I.\mathit{stopWindow} = \mathit{DT\_stopWindow\_Outside};\}\end{aligned}$$

System output *trainDoors* corresponds to subsystem output *trainDoorRelease* (O_{11}). System output *psdDoors* cannot be mapped directly to subsystem output *psdRelease* (O_{12}) as the interface partners differ.

We observe the two different types of refinement operations made in our example:

1) *Interface refinement:* The system requirement input *stopWindow* is refined to subsystem requirement inputs *trainPos*, *stopPoint* with *stopWindowPlus* and *stopWindowMinus* tolerances, as depicted in Fig. 8.
2) *Decomposition:* The subsystem requirement assigns the calculation of the door release conditions to the on-board subsystem. This is an architectural decision (which may be mandatory due to architectural constraints).

D. Restructuring the Requirements

Goal of our refactoring approach is to close the gap between abstract system and detailed subsystem requirements by creating a detailed system requirement that contains only the interface refinement on system level of our subsystem requirement *doors release*.

In a first step, we identified the relevant subsystem requirements with subsystem communication (for train control level C):

- *Doors release:* The on-board subsystem calculates rules for train door and PSD authorization.
- *Train doors release command:* The on-board subsystem communicates train door release command to an external system (that releases the train doors).
- *Authorize PSD opening command:* The on-board subsystem communicates the PSD authorization command to the wayside subsystem.
- *Doors open:* The on-board subsystem calculates rules for train door and PSD opening if the train doors are released.
- *Train doors open command:* The on-board subsystem communicates the train door open command to an external system (that opens the train doors).

TABLE I
DATA TYPES USED FOR THE FORMALIZATION OF THE SYSTEM REQUIREMENTS

Variable	Data type	Values
trainControlLevel	$DT\_TrainControlLevel$	$DT\_TrainControlLevel\_A$
		$DT\_TrainControlLevel\_B$
		$DT\_TrainControlLevel\_C$
trainMovingStatus	$DT\_TrainMovingStatus$	$DT\_TrainMovingStatus\_Moving$
		$DT\_TrainMovingStatus\_Standstill$
stopWindow	$DT\_StopWindow$	$DT\_StopWindow\_Outside$
		$DT\_StopWindow\_Within$
trainDoors, psdDoors	DT_Door	DT_Door_Close
		DT_Door_Open

TABLE II
DATA TYPES USED FOR THE FORMALIZATION OF THE SUBSYSTEM REQUIREMENTS

Variable	Data type	Values
trainControlLevel	*DT_TrainControlLevel*	see Table I
trainMovingStatus	*DT_TrainMovingStatus*	see Table I
trainDoorRelease, psdRelease	*DT_DoorRelease*	*DT_DoorRelease_Open*
		DT_DoorRelease_Close
trainPos, stopPoint, stopWindowMinus, stopWindowPlus	*int*	

- *PSD open command:* The on-board subsystem communicates the PSD open command to the wayside subsystem.
- *PSD open rule and command:* The wayside subsystem communicates the PSD open command to an external system (that opens the PSD) if the door open authorisation is set and as soon as the PSD open command is received.

The resulting merged requirement is:

In train control level C:
The on-board subsystem calculates rules for train door and PSD authorization. The on-board subsystem communicates the train door release command to an external system (that releases the train doors). The on-board subsystem communicates the PSD authorization command to the wayside subsystem.
The on-board subsystem calculates rules for door opening if the train doors are released. The on-board subsystem communicates the train door open command to an external system (that opens the train doors). The on-board subsystem communicates the PSD open command to wayside subsystem. The wayside subsystem communicates the PSD open command to an external system (that opens the PSD) if the door open authorization is set and as soon as the PSD open command is received.

The merged requirement is *generalized* to the system level by removing all internal communication information in a second step. The resulting detailed system requirement summarizes all subsystem requirements on train door and PSD door authorization and open commands.

The resulting detailed system requirement called *detailed doors management* is:

In train control level C:
The TGMT calculates rules for door authorization. The TGMT communicates the train door release command to an external system (that releases the train doors).
The TGMT calculates rules for door opening if the train doors are released. The TGMT communicates the train door open command to an external system (that opens the train doors). The TGMT communicates the PSD open command to an external system (that opens the PSD).

Denoting the outputs of *doors management* with O and of *detailed doors management* with O' the representation function is:

```
$O.psdDoors = O'.psdDoors;$
If ($O'.trainDoorRelease == DT\_DoorRelease\_Open()$ &&
  $O'.trainDoors == DT\_Door\_Open()$){
  $O.trainDoors = DT\_Door\_Open();$ }
else{
  $O.trainDoors = DT\_Door\_Close();$ }
```

E. Findings in the Case Study

In total the case study contained 11 system requirements and 28 subsystem requirements that were in scope of our project (note that we only modeled the PSD part of TGMT). Out of the 28 subsystem requirements there were 19 that contained both interface refinement and decomposition information. For these 19 subsystem requirements we introduced 15 detailed system requirements in the model. They were extracted from the subsystem requirements and contained the interaction refinement information of the system interface as provided by the subsystem requirements. We formalized 39 out of 54 requirements by applying the method described in Section III.

By formalizing the requirements we identified a conflict between two requirements on the on-board subsystem as both requirements defined a concretization of the same input value *stopping window* specified in a system requirement. In such a situation the requirements can bei either contradicting - in this case we would have discovered a flaw in the requirements - or complementing each other. As our case the requirements were not contradicting but complementing each other, we could resolve the conflict by integrating both requirements.

Setting up formal refinement specifications revealed flaws in requirements (e.g. inconsistencies or incomplete coverage of higher level requirements by lower level requirements). By introducing detailed system requirements we were able to specify a formal notion of tracing and refinement from system requirements to the architecture specification.

The case study gave a strong argument for model-based requirements engineering: Even when high quality input specifications are available, as it was the case in our study, model-based requirements engineering revealed discrepancies and flaws, which in turn could be resolved by our approach.

VII. Summary and Outlook

In an exploratory case study provided by Siemens, we analyzed the refinement from system to subsystem requirements. For the analysis we used the model-based requirements engineering approach MIRA that allowed us to precisely define the refinements of requirements. System requirements described the desired system capabilities and properties on an abstract level. Subsystem requirements had a scope on subsystems, while at the same time detailing the system interfaces.

One of the main results of the case study was that the resulting formal refinement specifications between the given system and subsystem requirements may become very complex. An analysis of the refinement revealed that the complexity results from the fact that two refinement operations, namely *interface refinement* and *decomposition*, were performed in one step. By *interface refinement*, we understand a refinement of requirements on the system interface (without adding architectural decisions). For example, the system requirement defines an abstract *stopping window* in which the train has to stop. The subsystem requirement defines detailed interval boundaries for the stopping window. By *decomposition* of system requirements to subsystems, we refer to the breakdown of requirements to subsystems (without changing the system interface). For example, a subsystem is defined which calculates the fulfillment of the stopping window. We call the inclusion of both refinement operations in one refinement step a "gap" between system and subsystem requirements. A gap between system and subsystem requirements affects the quality assurance of the refinement and hinders reuse of information on the system level.

To close the gap, we propose to specify a new level of requirements called *detailed system requirements* that have a scope on the system, but contain the interface refinements of the subsystem requirements. We introduced a bottom-up restructuring approach that provides transformation operations for subsystem requirements in order to obtain the detailed system requirements. We discussed the introduction of detailed system requirements. First, the number of requirements and refinements between requirements increase due to the new level of requirements. On the other hand, refinement specifications are simplified as every refinement specification only specifies the interface refinement or the decomposition. By documenting the interface refinement on system level, it is possible to reuse this information, when the system decomposition changes. Quality assurance of the refinement on system level can now be performed on system level and independently from the system decomposition.

We demonstrated our approach using examples of the Siemens case study and presented the findings of the application of the approach in the case study.

In our case study, some of the detailed system requirements were composed of numerous subsystems requirements. This led to bloated requirements, in terms of extensive input and output port definitions. This points to the (as yet) open research question regarding the adequate size of detailed system requirements most suited for formal refinement specification. Furthermore, we will extend the restructuring approach in order to obtain requirements of adequate size, e.g. by introducing new operations such as splitting requirements according to system behavior. We will also be looking into possibilities for automating the restructuring approach, with a focus on scalability.

References

[1] W. Böhm, M. Junker, A. Vogelsang, S. Teufl, R. Pinger, and K. Rahn, "A formal systems engineering approach in practice: An experience report," in *SER&IPs*, 2014, pp. 34–41.

[2] S. Teufl, D. Mou, and D. Ratiu, "Mira: A tooling-framework to experiment with model-based requirements engineering," in *RE*, 2013, pp. 330–331.

[3] M. Broy and K. Stølen, *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer, 2001.

[4] M. Broy, "A logical basis for component-oriented software and systems engineering," *Comput. J.*, vol. 53, no. 10, pp. 1758–1782, Dec. 2010.

[5] F. Hölzl and M. Feilkas, "Autofocus 3 - a scientific tool prototype for model-based development of component-based, reactive, distributed systems," in *Model-Based Engineering of Embedded Real-Time Systems*. Springer, 2010, pp. 317–322.

[6] A. Kondeva, D. Ratiu, B. Schätz, and S. Voss, "Seamless model-based development of embedded systems with af3 phoenix," in *ECBS*, 2013, pp. 212–212.

[7] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*, 1999, pp. 411–420.

[8] J. O. Blech, D. Mou, and D. Ratiu, "Reusing test-cases on different levels of abstraction in a model based development tool," in *MBT*, 2012, pp. 13–27.

[9] B. Penzenstadler, "Exactly the information your subcontractor needs: Desyre; decomposing system requirements," in *RePa*, 2011, pp. 1–10.

[10] A. van Lamsweerde, "Goal-oriented requirements engineering: a guided tour," in *RE*, 2001, pp. 249–262.

[11] N. Jazdi, C. Maga, and P. Göhner, "Reusable models in industrial automation: Experiences in defining appropriate levels of granularity," *IFAC*, pp. 9145–9150, 2011.

[12] M. Broy, "Multifunctional software systems: Structured modeling and specification of functional requirements," *Science of Computer Programming*, vol. 75, no. 12, pp. 1193–1214, 2010.