TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

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

SPES 2020 Deliverable D1.2.B-9

Managing the Complexity of Interfaces

Current Challenges

Software Plattform Embedded Systems 2020

Author:	Daniel Ratiu
	Judith Thyssen
	María Victoria Cengarle
	Alexander Harhurin
Version:	1.0
Date:	April 15, 2010
Status:	Released

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

About the Document

The decomposition of a system into smaller sub-systems is a fundamental means to reduce the complexity of the system in a “divide and conquer” fashion, to moreover distribute the labor between different development participants and between integrators and suppliers, to enable reuse of components, diagnosis of sub-systems, and testing of sub-systems, etc. The definition of clear interfaces between sub-systems is a prerequisite for the sucess of the decomposition. Interfaces represent the black-box specification of a system that focus only on the usage information while abstracting away from implementation details.

In this document we discuss the challenges posed by the definition of complex interfaces. Following this, we briefly sketch solutions offered by model-based development approaches to systematically approach these challenges and to ease the definition of interfaces.

This document is realized in the frame of the SPES2020 project and the collaboration between the ZP-AP1 and AWP-AV.

1 Introduction

Interfaces represent the borders of a system. Definition of interfaces is the most important means to specify the black-box view over the system. Interfaces abstract away from implementation details and represent the pure capabilities of the system that are necessary for using the system and integrating it in any context. The information provided by an interface must be enough for engineers to rely exclusively on that information in order to safely use the system specified by that interface. Interafces specifications play a central role to realize the contract between suppliers and integrators.

2 Challenges in working with complex interfaces

Working with complex interfaces as they can be found in avionic systems raises various challenges.

Richness of interfaces One of the key question in dealing with interfaces is to determine what information need to be integrated in the interface specification. On the one hand, the interface specification should be rich enough in order to capture all relevant details that allow the integration of the component with the rest of the system. On the other hand, the interface specification should abstract from all implementation details in order to remain simple and easy to understand even for complex systems.

Typically, there is a wide variety of information that is needed when specifying a rich interface. Typical elements of an interface specification are the following:

- **identifier** – the interface should have a unique name in the system so that it can be addressed and traced through the development.
- **data types** – besides the name of the interface, we need also information about the types of data exchanged by the interface. Data types have two main functionalities: firstly, they are indispensable means to structure the information and reduce the complexity; and secondly, different kinds of logical information should be assigned to different (incompatible) types in order to avoid the misuse of the interface.
- **attributes** – e.g. periodicity, measurement/unit,...
- **black box behaviour** – offers information about the behaviour of the system that implements the interface. We consider that the real-time requirements belong to the behavior of the system.

Support for complex analyses Depending on the level of detail of the interface specification, different analyses can be performed. An open question is to decide what information must be captured by the interface specification in order to enable what kind of analyses. Below typical analysis criteria are listed that are usually performed on interfaces:

- **Compatibility:** Given two systems described through their interfaces, can we identify whether both systems can be composed into a bigger system? We make a difference between
  - *syntactic compatibility* – meaning that the data types are compatible with each other, and
  - *behavioural compatibility* – meaning that the behaviour of the two systems are composable.
- **Refinement:** Given two interfaces, does one represent the refinement of the other?
- **Compliance:** Given a system implementation (e. g., as a state machine or a C program) and the definition of an interface, is the system implementation compliant with the interface specification?

These analyses are needed not only during the construction of the system but also during the implementation of changes, be these modifications to the interface or to the implementation of the system or any of its components.

Support for implementation of changes Due to new system requirements interfaces have to be changed. While performing an interface change it is important to estimate which other parts of the system need to be changed. Thus, concepts are needed that allow to perform impact analyses based on changes of a given interface specification.

Support for handling the complexity of large interfaces Complex systems have sub-systems and thereby many interfaces. To deal with complex interfaces of large systems imposes various challenges:

- **Concepts to reduce complexity.** What suitable concepts to handle the complexity of interfaces of real-time systems (e. g., compositionality/modularization).
- **Methodological support.** Methodological steps for a disciplined definition of interface specifications by focusing in each step only on relevant details and incrementally adding more information as it is needed.
- **Best practices.** Establishing and checking guidelines/best practices for description of interfaces that address questions like: are interface specifications easy, intuitive to read/write? do the interface specification reflect the intended meaning of the system? are the interfaces usable (i. e., easy to use, hard to misuse)?
- **Notation/Graphical representation.** Which notations are adequate for different tasks that we need to do with interfaces? How can interface specifications be graphically represented? Hereby, the scalability of the notation and its adequacy for specific tasks are essential.

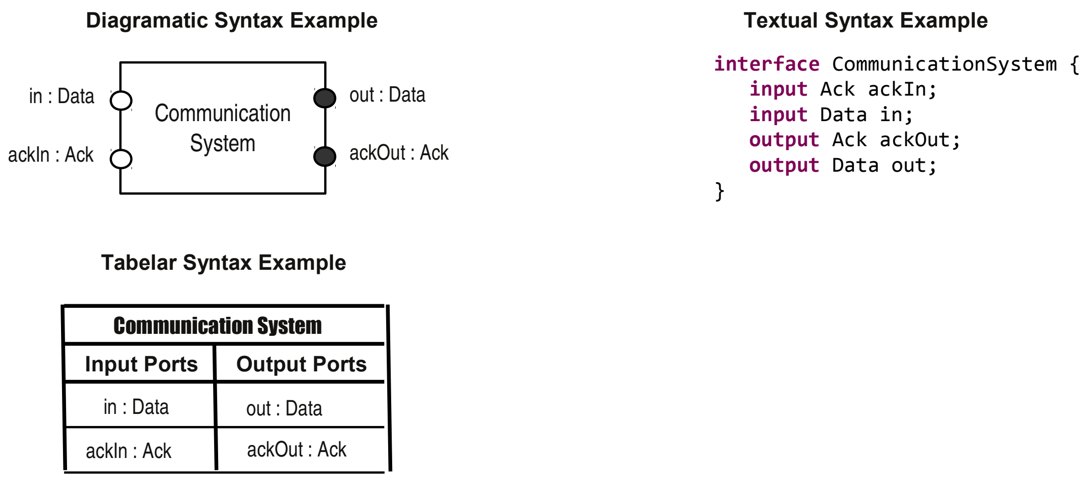
3 Possible Working Directions

The SPES2020 project has in focus the use of model based development for increasing the quality and productivity of the development of embedded systems. In ZP-AP1, we have in focus the following topics that address (part of) the above mentioned challenges.

3.1 Notation vs. Abstract Syntax

Different tools used to define interfaces offer different notations. Behind each notation is a set of constructs that represent the abstract syntax (concepts) of the language. These concepts tell us the powerfulness of the modeling language. The notations are however very important for the practibility of the language use and they offer support for performing different tasks.

In Figure 1 we present three typical notations (concrete syntaxes) for describing interfaces: a diagrammatic notation, a textual notation and tabular notation. Each of these concrete syntaxes, even if they represent the same abstract concepts, have advantages in different situations: for example the diagrammatic notations can be animated during simulation, the tabular notations are especially compact, and the textual notations ca support auto-completion functionality when interfaces are built.



```
interface CommunicationSystem {
   input Ack ackIn;
   input Data in;
   output Ack ackOut;
   output Data out;
}
```

Communication System	
Input Ports	**Output Ports**
in : Data	out : Data
ackIn : Ack	ackOut : Ack

Figure 1: Examples of notations for describing interfaces

To sum up, we envision firstly to define an adequate language for specifying interfaces and define different views in form of concrete notations that address different tasks. The language should allow us to directly capture all the information that is needed for specifying the interface. In the SPES2020 deliverable D1.2.B-1 we describe in detail the differences between concrete syntax, abstract syntax, and semantics.

3.2 Levels of specification

The interface specification of a system or component comprises syntactical information, i. e., the input ports and the corresponding data types and attributes. Additionally, the interface specification can be enriched by behavioural aspects. A formal definition of syntactic and semantic interface is given in [HHR09] (available also as the SPES2020 deliverable D1.1.A-1).

3.2.1 Syntactical Interfaces

The syntactical interface of a component consists of a set of input and output ports. Each port has a name and a type (e. g., `Bool`, `Int` etc.). These ports are the exclusive communication points of a component with its environment. In Figure 1, examples for the graphical representation of the syntactic interface of a component are given.

To address the problem of the "richness of interfaces", we use a "data dictionary", where complex data types and relations between them (like refinement, inheritance or "includeIn") can be defined.

The data dictionary may be compared to functional languages, well known in computer science. These languages usually define a set of predefined data types and allow to define more complex data types, according to the developers needs. Predefined data types are e. g., `boolean`, `int` or `string`. Some basic arithmetic functions like addition or multiplication should also belong to the predefined data types.

Complex data types are created in a data dictionary based on the predefined types. A complex type usually consists of constructor and selector functions. A constructor creates a type based on other types. For example, to create a type `position`, which consists of three `real` values, a creator $crPosition(x : Real, y : Real, z : Real)$ may be applied. Selectors define the access to a complex data type. For example, to get the altitude of the `position`, we define $getAltitude(crPosition(x, y, z)) := x$.

Functions on the data types may be arbitrarily complex. For example, the function $collision(p1 : Position, p2 : Position) : Boolean$ may be defined as follows:

$$collision(crPosition(x1, y1, z1),\ crPosition(x2, y2, z1)) := (x1 = x2\ \&\ y1 = y2\ \&\ z1 = z2).$$

3.2.2 Interface Behaviour

In addition to the syntactic information, the interface specification can be enriched by behavioural aspects describing the interface behaviour that can be observed at the boundaries of a component/system. It captures behavioural restrictions/requirements. Exemplary, behavioural extensions of the interface specification could be used to describe (simple) protocols. Formally, the interface behaviour is given as a relation between input and output messages. It restricts the set of all syntactically valid input/output combinations to a set of semantically valid behaviours.

Interesting research questions concerning the interface behaviour that could be addressed within SPES are:

- Selection of properties that can be annotated as behavioural specification.
- Suitable formalism to denote interface behaviour: assumptions/guarantee, interface automata, logical formula.
- Analyses that can be performed based on behavioural interface specifications (cp. Section 3.5).

3.3 Basic Concepts of the Underlying Theory

In order to be able to deal with complex interfaces, the underlying theory must provide different properties. This applies for the syntactic as well as the semantic interface.

- **Modularity and Compositionality:** Compositionality is the property of a theory that allows to deduce the interface behavior of a system from the interface behavior of its subsystems. It is the basis for the incremental and modular development of (sub-)systems. Modular development of sub-systems reduces the complexity through "divide and conquer" – instead of building monolithic systems in one step, compositionality enables to incrementally build systems out of modularly specified parts. Besides, compositionality is essential for the reuse of existing components and its interface specifications.
- **Abstraction and Refinement:** Refinement enables the transformation between a more abstract interface specification into a more concrete one without loosing the properties of the former one. The concept of refinement/abstraction allows us to start with high-granular descriptions and to incrementally refine them into more detailed ones. Refinement is especially useful in relating interface specifications from different abstraction levels: if the more concrete interface specification is a refinement of a more abstract one, we are sure that the refining interface specification guarantees all the properties of the abstract one.

For more details see [HHR09].

3.4 Methodology – Towards a disciplined definition of interfaces

The models developed during the system development can be seen at different levels of abstraction. Abstraction means generally leaving out the unnecessary details. We achieve abstraction by following two strategies:

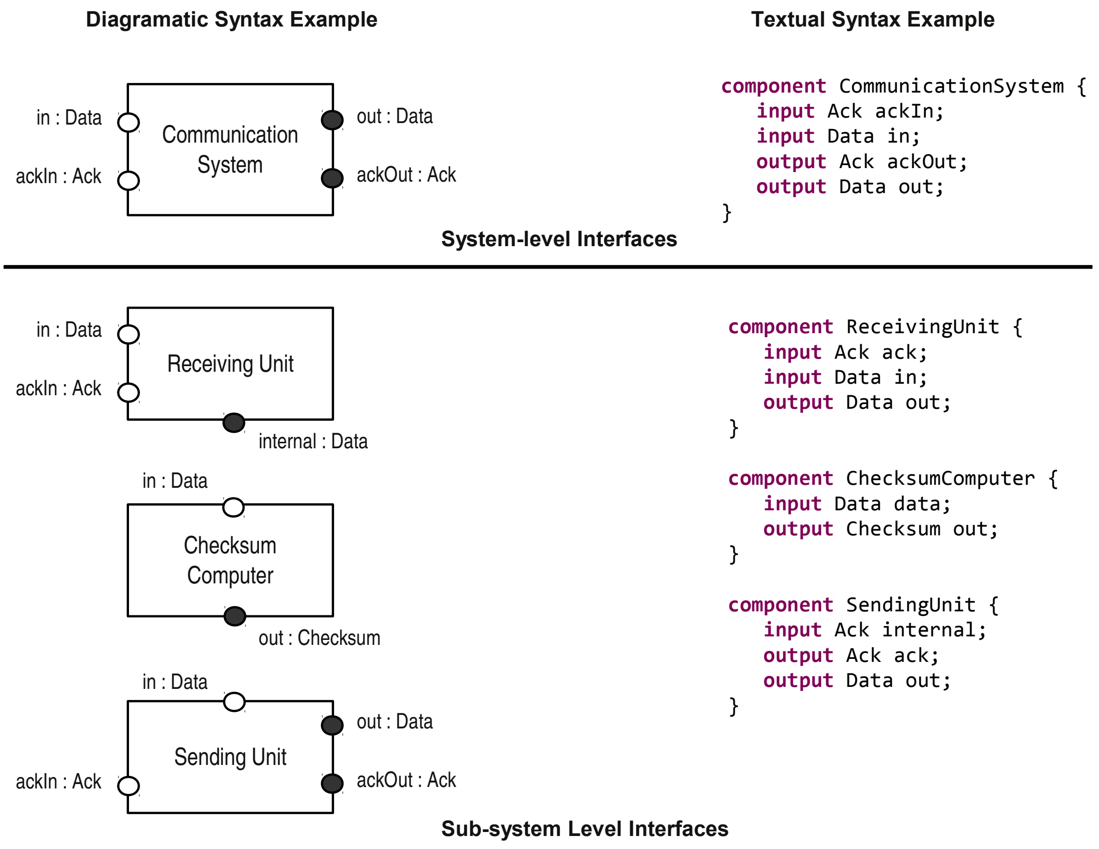
- consider the system at **different granularity levels** (through a magnifier) and leave out the information that is not necessary at the current granularity level. From the point of view of the granularity level, we distinguish between system-interfaces, sub-system interfaces, sub-sub-system interfaces and so on up to the interfaces of atomic components. A lower level of granularity makes the interfaces defined at a higher level more explicit and introduces other elements in the interface that represent the implementation details.

- at each granularity level, consider the system from three points of view: **user functionality**, **logical architecture** and **technical implementation**. Corresponding to each of this point of view we have specific interfaces: the user functionality interfaces define interfaces that address the users (e. g., by using abstract types), the logical interfaces define the interfaces of the system by considering more complex details like the exact communication protocol or the existence of error recovery mechanisms (e. g., checksums), and at the technical level we describe the order of sent bits and their exact interpretation (e. g., int32, int16).

A more detailed introduction in our framework of abstraction layers can be found in [RST09].

It is important that the relation between abstract interfaces and how they are implemented by more concrete interfaces to be explicit.

Example: In Figure 2 we present an example of the definition of system interfaces at two levels of granularity: system level and sub-system level.



```
component CommunicationSystem {
   input Ack ackIn;
   input Data in;
   output Ack ackOut;
   output Data out;
}
```

```
component ReceivingUnit {
   input Ack ack;
   input Data in;
   output Data out;
}

component ChecksumComputer {
   input Data data;
   output Checksum out;
}

component SendingUnit {
   input Ack internal;
   output Ack ack;
   output Data out;
}
```

Figure 2: Example of a communication system: system view, sub-system view

3.5 Analyses of interfaces

The term interface refers to an abstraction that a module (e. g., sub-system, software component) provides of itself to the outside thus separating external communication from internal operation. This dissociation into interface and internals allows modification of the latter that is transparent to the communication partners of the entity.

3.5.1 Compatibility of interfaces

Two different interfaces (or parts thereof) can be composed giving rise to a more complex interface. Compatibility criteria help in deciding whether or not two interfaces can be composed; in case they are, the interfaces are said to be compatible. In Figure 3 two exemplary interfaces are shown that are, from a behavioural point of view, incompatible with each other: the assumption of the divider, namely that the divisor is not zero, is not guaranteed by the output of the adder component.

```
// interface of a component that
// implements addition a+b

interface Adder {
   input Integer a, b;
   output Integer c;
}
```

```
// interface of a component that
// implements division a/b

interface Divider {
   input Integer a, b;
   output Integer c;
   Invariant:
        b * c <= a
   A/G:
        A: b != 0
}
```

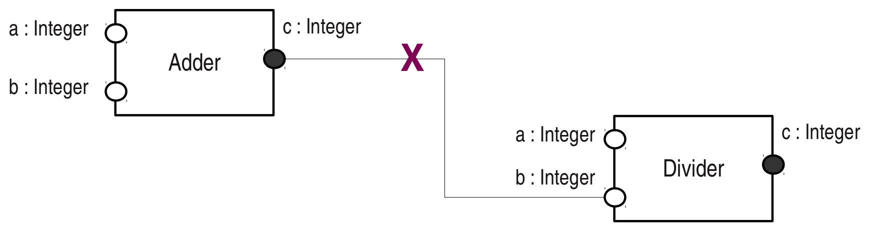


Figure 3: Examples of (semantic) incompatibility between two interfaces

3.5.2 Refinement of an interface by another interface

An interface is said to refine, or to be a refinement of, another one if the former, called the *finer* interface, can be utilised whenever the latter, called the *coarser* interface, is expected. In case the interfaces describe functions, the well-known criterion of contra-/co-variance is typically used, meaning that finer interface inputs values of supertypes and outputs values of subtypes of the respective types of the coarser interface. If the behaviour of these functions is moreover detailed by means of, e. g., an invariant, then the invariant of the finer interface must imply the invariant of the coarser interface. If the behaviour of those functions is detailed by means of a pair assumption/guarantee, then, on the one hand, the assumption of the finer interface must imply the assumption of the coarser interface and, on the other, the guarantee of the coarser interface must imply the guarantee of the finer interface. In Figure 3 we present an example of two interfaces that implement the functionality of an adder. The interface "Adder2" is a refinement of the interface "Adder1" since its input ports are more general and the specification of Adder2 implies the specification of Adder1.

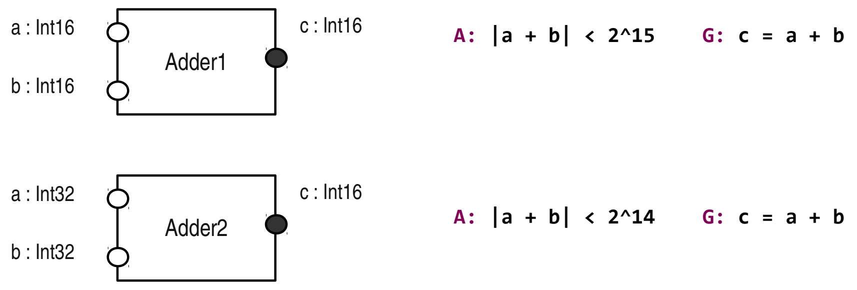


Figure 4: Examples of refinement between two interfaces (Adder2 refines Adder1)

3.5.3 Compliance of an implementation with an interface

The abstraction provided by interfaces can be purely syntactic, e. g., a list of ports with corresponding types, or more sophisticated and allow a description of the entity's behaviour in form of, e. g., invariants or pairs of assumption/guarantee. Even more, an interface can have an inherent notion of state, meaning that the behaviour showed by its implementation depends on the state in which the entity is located. In any case, the question arises whether an implementation complies with its interface; if not only syntax is specified by the interface, then the verification of compliance is more complex.

In Figure 5 we present an example of an interface and four components, represented as C functions, that implement this interface. Besides the types and the number of inputs and outputs, the interface specifies also behaviour in form of an invariant and a pair assumption/guarantee. From the four implementations, only the second one complies the interface specification. The first implementation violates the invariant, the third and fourth are not well-typed.

It turns out that the concepts of compliance and refinement are related. Indeed, if an interface I_1 is refined by an interface I_2, and a program P_2 complies with I_2, then P_2 also complies with I_1. Additionally, the concept of refinement can be broaden to span also over programs. Hence, if a program P_1 complies with an interface I_1 and is refined by a program P_2, then P_2 also complies with I_1.

The relations of refinement and of implementation should be congruent with the composition operator: if an interface I is composed by I_1 and I_2, what may be denoted by $I = I_1 \oplus I_2$, and moreover I_1 is refined by I_1', then it is desirable that I be refined by $I' = I_1' \oplus I_2$. And similarly for implementation: if $P = P_1 \oplus P_2$ complies with I and P_1 is refined by P_1', then $P' = P_1' \oplus P_2$ likewise complies with I. Note that here we simply use and do not detail an operator $\oplus$, which may be undefined for certain pairs of interfaces or of programs, and may need additional operands; for instance, additional information may be the output and input

```
// interface of a component that implements division a/b

interface Divider {
   input Integer a, b;
   output Integer c;
   Invariant:
        b * c <= a
   A/G:
        A: b != 0
}
```

```
int Divider1(int a, int b) {
   int c = 0;
   do {
     c++;
     a = a - b;
   } while(a > 0)
   return c;
}
```

```
int Divider2(int a, int b) {
   return a / b;
}
```

```
int Divider3(char a, char b) {
   return a / b;
}
```

```
float Divider4(int a, int b) {
   return a / b;
}
```

Figure 5: Examples of an interface specification and different implementations as C-like functions (only the second one is compliant)

ports of two interfaces that are to be connected, assuming a certain type compatibility, and perhaps also a channel name.

References

[HHR09] Alexander Harhurin, Judith Hartmann, and Daniel Ratiu. Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems. Technical Report TUM-I0924, Technische Universität München, 2009.

[RST09] Daniel Ratiu, Wolfgang Schwitzer, and Judith Thyssen. A System of Abstraction Layers for the Seamless Development of Embedded Software Systems. Technical Report TUM-I0928, Technische Universität München, 2009.