13 Components and Contracts

13.1 Introduction

One of the key desiderata in component-based development for embedded systems is the ability to capture functional and extra-functional properties in component interfaces, and to verify and predict corresponding system properties. For real-time systems, this is perceived to be particularly important for properties such as timing and quality-of-service.

In this section, we review existing techniques for capturing, verifying, and predicting different properties of component and system behaviour. Properties of components can be expressed in their contracts, hence the title of the section. The term contract can very generally be taken to mean “component specification” in any form.

A contract is in practice taken to be a constraint on a given aspect of the interaction between a component that supplies a service, and a component that consumes this service. Component contracts differ from object contracts in the sense that to supply a service, a component often explicitly requires some other service, with its own contract, from another component. Therefore the expression of a contract on a component-provided interface might depend on another contract from one of the component-required interfaces. For instance, the throughput of component A doing some kind of computation on a data stream provided by component B clearly depends on the throughput of B.

It is indeed challenging to develop a practical framework for reasoning about complex component properties (e.g., performance properties) stated in contracts, e.g., to infer global system (performance) properties. A complete solution to this problem requires powerful mathematical reasoning, e.g., about properties of stochastic processes. A pragmatic, more modest, approach to this problem, which does not need powerful mathematical reasoning, is to agree on a small set of fixed contracts, or a small set of fixed building blocks for contracts. For each contract, one can then in advance develop techniques for monitoring or verifying that component implementations satisfy the contract, and techniques for inferring system properties from component contracts. For instance, for performance properties, one can define a fixed set of different levels of performance, and for each level define rules for run-time monitoring and for component interoperability.

In simple cases, such a scheme can be seen as constructing a type system for specifying properties. More complex cases may involve constraints expressed in some type of logic, and thus checking beforehand that components interact correctly then need some form of theorem proving techniques.

To structure the exposition into different types of component properties, we use the classification of contracts proposed by Beugnard et al. [BJP99], where a contract hierarchy is defined consisting of four levels.

- **Level 1**: Syntactic interface, or signature (i.e. types, fields, methods, signals, ports etc., that constitute the interface).
• **Level 2**: Constraints on values of parameters and of persistent state variables, expressed, e.g., by pre- and post-conditions and invariants.

• **Level 3**: Synchronization between different services and method calls (e.g., expressed as constraints on their temporal ordering).

• **Level 4**: Extra-functional properties (in particular real-time attributes, performance, QoS (i.e. constraints on response times, throughput, etc.). We will separate this level into two aspects
  - **4a**: timing properties (e.g. absolute time bounds)
  - **4b**: Quality of Service properties, typically given by performance measures, often formulated in stochastic terms (e.g. average response time).

Currently, most component models support only level 1 contracts, while some models support also other levels (see section 14). In the remainder of section 13, we will survey techniques for capturing and reasoning about component and system properties, discussing each aspect separately. We will use the four levels of the Beugnard hierarchy for structuring our treatment of different interface properties. Regarding level 4, we make a separation between timing properties (e.g. absolute time bounds) and stochastically formulated performance properties (e.g. average response time). In addition, we briefly treat reliability properties.

For each aspect, we will consider techniques for

- expressing properties of systems and components,
- predicting or verifying system properties from component properties, in particular for doing this statically at design-time,
- checking that component properties are compatible (assumptions made in one component specification are guaranteed by some other component specification),
- verifying that component implementations satisfy properties given in component specifications, and
- compile-time and run-time support for enforcing system or component properties.

### 13.2 Level 1 – Syntactic Interfaces

**Definition**

By a *syntactic interface*, we understand here a list of operations or ports, including their signatures (the types of allowed inputs and outputs), by means of which communication with a component is performed.

Generally speaking, a *type* can be understood as a set of values on which a related set of operations can be performed successfully. Belonging to a given type usually implies constraints that go beyond what value is denoted exactly, most notably *how* the value is stored (required when operations are performed). Once types have been defined, it is possible to use them in specifications of the form: if some input of type \( t_{in} \) is given, then the output will have type \( t_{out} \).

*Type safety* is the guarantee that no run-time error will result from the application of some operation to the wrong object or value. A *type system* is a set of rules for checking type safety (a process usually called type checking since it is often required that enough information about the typing assumptions has been given explicitly by the