1. Introduction

This position paper describes a component oriented approach for the modeling of the regulator case. The modeling method is one of the results of the SEESCOA project, sponsored by the Flemish IWT and 6 industrial partners. In this project we investigate the development of embedded software using high-level software concepts.

The SEESCOA method is in part based on a combination of Component Based Development and Contract Based Development (see [1]). Important keywords are: component, port, connector, interface and contract. Some of these concepts are also present in UML for Real-Time and ROOM ([2] and [4]).

The presented notation and method is targeted to embedded software, where a developer has to take into consideration typical embedded software issues, like timing and memory usage. In this paper we focus on timing, which is also an important aspect of the case.

The presented concepts will be made clear in underlying examples, based on the case.

2. Modeling: basic concepts

An application is constructed by connecting components. Components exchange messages via connectors. Every message is sent asynchronously, and can contain parameters. These constructs are used to model the logical part of the application.

To enable the exchange of messages between two components a protocol is needed: this is the “language” spoken by both components. An important part of this protocol are Timed MSC’s [5]. Using these Timed MSC’s it is possible to perform static timing verifications.

Also, it is possible to model non-functional requirements by using contracts. These contracts can be compiled to code, enabling the runtime monitoring and scheduling of the contracts. This corresponds to dynamic verification. A timing contract is a particular type of contract: it models timing in an application.

3. Regulator Case: Logical View

In the left figure two components are shown which are used to compute and display car speed: speed display and speedometer. Because the speedometer has to send speed updates to the speed display, it needs a port which is connected via a connector to a port on the speed display component.

Ports have a multiplicity: this indicates how many ports of this type exist. Every port also has a role. The role of a port indicates which role the port plays in the communication between both components.

Most of the concepts presented in this figure have similar counterparts in the ROOM method.

The ports of speedometer and speed display have to “speak” the same protocol. This protocol is also called the interface or type of the port and is specified at 4 different levels:

- **Syntactic Level**
  - Description of the signatures of the messages.
- **Semantic Level**
  - Description of pre- and postconditions.
Synchronization Level
Description of the sequence of messages, loops, alternative paths, and so on. To describe synchronization we use Timed MSC’s. One can compare these to Sequence Diagrams used in UML, where some extensions have been made to:
- Indicate loops (LOOP construct)
- Indicate optional paths
- Annotate timings

QoS Level
On this level one is able to describe the non-functional requirements and provisions of a component. In fact, the timing information of Timed MSC’s can also be seen as part of this level.

In the remainder of the paper we will only use the syntactic and synchronization level to annotate components.

Given this notation it is possible to construct instance models: an instance model is a view on a coherent part of the application. Considering the regulator case there will be three such instance models: a case speedometer model, a case regulator model and a case integration model.

3.1. Case Speedometer Model

The model presented below shows three connected components: speedometer uses a clock to update the speed at a frequency of 2 Hz and also to send out these updates to interested parties, like the speed display. The model also contains information about the ports being used: their name, and their syntactic and synchronization properties. To configure the clock a clock ctrl port is used. The clock sends out ticks() via the clock tick port to the speedometer. The speedometer then sends out updates to speed display via its speed update port.

3.2. Case Regulator Model

In the next instance model the case regulator is described. A part of this case regulator is the regulator component. This is a reusable component that continuously calculates δC. The regulator can be in three states: ON, OFF and STDY. The state of the regulator can be read via a regulator status readout port (having multiplicity greater than one). The regulator can be controlled by means of a regulator control port. The syntactic and synchronization interface of this last port is shown in the figure on the right. One can deduce from the MSC that a ON() message should be sent first, followed by zero or more STDY() messages (indicated with the LOOP construct), and finally this sequence has to be terminated by sending out a OFF() message.

The regulator control port is used by a regulator controller component. The regulator controller offers in turn three ports: ON, STDY and OFF. These ports can be used by the monitor components. Every monitor component monitors specific events (like shutting down the engine = engine monitor, speed dropping below 50 km/h = speed monitor, …). The regulator controller is in charge of executing the ON, OFF and STDY commands sent out by these monitors. It also takes into account possible simultaneous occurrences of these commands. To conclude, there is also a regulator display used to display the state of the regulator to the car driver (ON, STDY or OFF).

1 Note the use of graphical sequence diagrams to show the interaction between two ports. Recall that communications are performed asynchronously, which means that a sender of a message doesn’t get blocked when sending a message.
The **regulator** also has a **regulator settings** port, used to set the regulation-frequency (2 Hz) and a **engine output** port, used to send out the computed δC’s. To read in the current speed a **speed input** port is used. This port will be connected to an available **speed update** port on the **speedometer**.

The figure below illustrates this schematically. Note that all **monitor** components are connected with the **OFF** port of the **regulator controller**. This means that every **monitor** can shut down the **regulator**. But not every **monitor** can power on the **regulator**; only the **on/off monitor** can do that.

### 3.3. Case Integration Model

The next model shows the components implementing the full functionality required in the case. The engine unit is a black box component, offering a **δC update** port used by the **case regulator** to post its computations and an **engine on/off status** port showing the state (on/off) of the **engine unit**.

### 4. Regulator Case: Non-Functional View

Besides connecting components one is also able to use **contracts**. This is done by creating **scenario** models. A scenario model describes a specific non-functional part of the model, and contains one or more contracts. A contract is an agreement between two or more components about a specific aspect of the application. Mostly these contracts have a non-functional meaning: they can represent an agreement about timings they have to adhere to.

These contracts have a specification purpose but do also have a meaning at runtime. Since the tool generates code for the models, it also generates code for **dynamic contract checking**. Contract violations (like **periodicity** or **worst case duration** violations) are logged or reported. As such, these contracts are used for runtime timing checks.
The timing contract presented in this paper is a TwoPartyTimingContract (TwoPartyTC). This type of contract is attached to two hooks (a start hook and an end hook). The contract is valid from the start hook till the end hook. Note that the start hook and end hook are moments in time: it can be the moment before sending a message, at arrival of a message or after the processing of a message. A TwoPartyTC is characterized by following parameters:

- **start hook**: start position of validity of the contract
- **end hook**: end position of validity of the contract
- **max period**: if this field has a value it indicates the periodicity of the actions
- **wcd**: if this field has a value it indicates the worst case duration of the actions

The figure on the right shows two timing contracts related to each other. They define the constraint: “The speed has to be displayed at 2Hz”. C1 specifies that the clock should send out a tick every 500 ms. C2 specifies that the speedometer should send an update to the display at 2 Hz and also that the display should process this update at 2 Hz. C1 can be seen as a contract the speedometer needs if it wants to guarantee C2. This is analogous to the real world situation where a supplier of a product/service is dependent on products/services of other suppliers.

The figure on the left shows a contract between two implicit connected ports. ‘Implicit’ means that a message sent out on the first port will result in the arrival of a different message on the second port. The contract shown here defines the constraint: “Shutdown response time for on/off switch actuation is 0.5s”. The contract is valid from hook a (before the event is sent out by the regulator on/off component) till hook b (after the OFF() message has been processed by the regulator component). Note: It is assumed that there is no time between the triggering of the off-switch by the cardriver and the sending of this event on the regulator on/off port.

5. Conclusion

The approach in this paper does not show a new method for modeling the functional part of an application since it is based on UML for Real-Time. However, UML for Real-Time does not have explicit constructs for annotating non-functional requirements (like timing and memory). In our approach we enable this by using a contract based method: timing requirements are put into contracts. These contracts have a specification meaning but also a runtime meaning (they are used to monitor timings in an application at runtime).

6. References


---

2 Note that the start and end hooks are the same here.
3 It is also possible to split contract C2 in two more detailed contracts.