An application of the DESS modeling approach: The Car Speed Regulator

Vieri del Bianco, Luigi Lavazza, Marco Mauri

CEFRIEL and Politecnico di Milano

Abstract

The ITEA Project DESS (Software Development Process for Real-Time Embedded Software Systems) aims at defining a methodology for the development of real-time software. The authors of this position paper are involved in the definition of methods for the specification and analysis phases. Our goal is to allow analysts to model real-time systems using an extension of UML, which can support the translation of models into formal notations. In this way we retain the expressiveness and ease of use of UML, while gaining the power of formal methods.

In this paper we employ our notation to model the proposed car speed regulator, in order to verify that such notation is able to represent the desired behavior of the system.

1. Introduction

UML [2, 3] provides a semi-formal graphical notation to represent different and often complementary views of software systems. However, the information contained in these diagrams is often imprecise, i.e., it lacks formal semantics. Moreover, UML does not support well real-time, embedded and resource-constrained system modeling, as it lacks a sound support for timing issues. The most well-known real-time extension of UML, UML-RT (based on ROOM methodology [4]), defines a very good architectural framework, but does not take into account time and constraints in general.

Our goal is to apply (as few as possible) extensions to UML and give it a precise semantic, in order to make it possible to translate UML diagrams into formal notations compatible with existent formal methods. The goal is to exploit formal methods in activities such as specification validation (e.g. via simulation or model checking) or test case generation.

More on the DESS approach can be found in [7].

In the rest of this paper we apply just one of the extensions of UML which have been defined in DESS: we employ the non-deterministic version of the `after` construct in UML state diagrams. In this extension, `after` accepts as an argument a time range, so `after([tLowerBound, tHigherBound))` means that a transition will happen after a time greater than or equal to `tLowerBound` and less than `tHigherBound`.

With this simple extension we are able to model the system described in [1] and the constraints the system behavior has to comply with. Constraints are expressed in an operational way, being embedded in the model. This is possible because the constraints are relatively simple. There are cases when this is not possible or not practical.

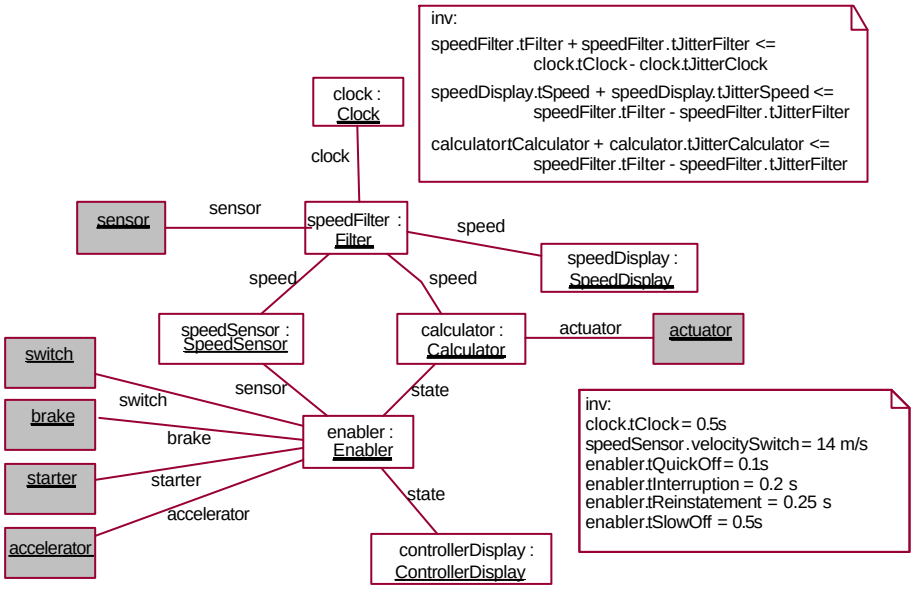

Figure 1: The object diagram describing the complete system (in gray the objects out of scope).

2. The Model

In this section we present the model of the system described in [1]. For space reasons we omit the class diagram. Instead an Object Diagram (Figure 1) shows the object instances and how they are connected with each other. We rely on the intuition of the reader to understanding objects attributes and methods.

The object diagram also includes constraints. Some of these are logical, involving different attributes of different objects in the system. Others are numeric: they assign specific values to constants (these are actually attributes whose value cannot be changed once the object is instantiated).

The following diagrams represent the behavior of every Class (and hence, instance) we introduced. The Clock (Figure 2) is very simple, and does not require a thorough explanation.

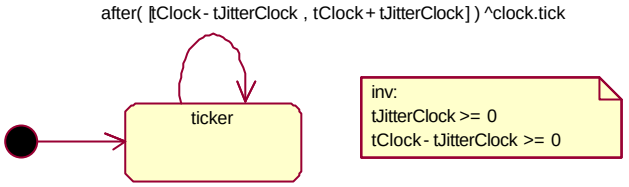

Figure 2: State Diagram of class Clock.

Note: the object diagram in Figure 1 contains several links having the same name (e.g., links labeled “speed”). This is actually a small syntactic extension, which allows the modeler to express message sending in a more compact way: for instance, when an object of type Filter (Figure 3) sends a message to the target “speed”, this actually means that the message is sent to all the objects reachable by means of links labeled “speed”. The name of the link is often used to identify the recipient of a message: for instance the clock (Figure 2) sends its message to the object connected via the “clock” link.

The behavior of the Filter and of the SpeedDisplay are described in Figure 3. Note that to let the filter loose no “ticks” of the clock we have to constrain the time response of the filter: this constraint is reported in Figure 1.

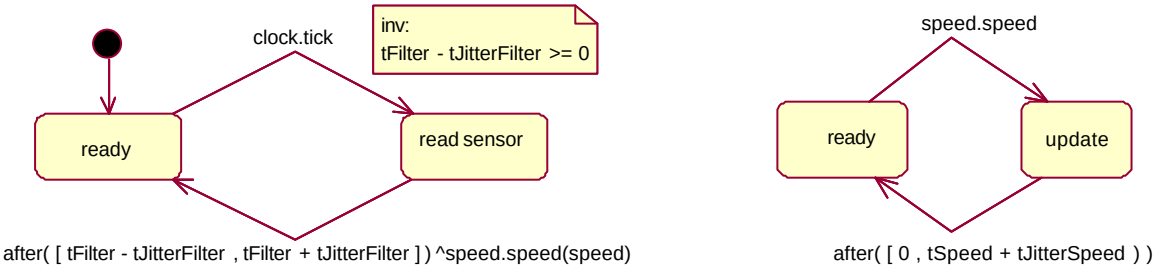

Figure 3: State Diagrams of classes Filter (left) and SpeedDisplay (right).

The Enabler (Figure 5) and the Calculator (Figure 4) are part of the regulator (in the design phase they could be represented as two parallel threads of the regulator). The main purpose of the enabler is to understand when to turn on and off the whole regulator, while the calculator computes the values to be passed to the actuator that controls the engine.

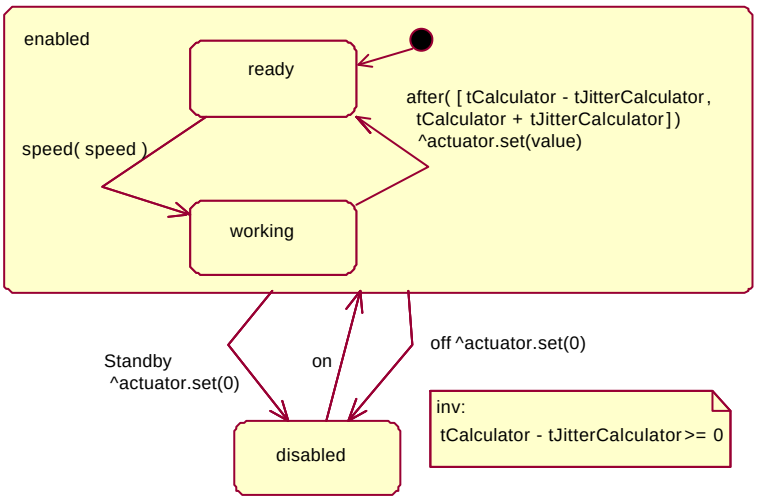

Figure 4 State Diagram of class Calculator

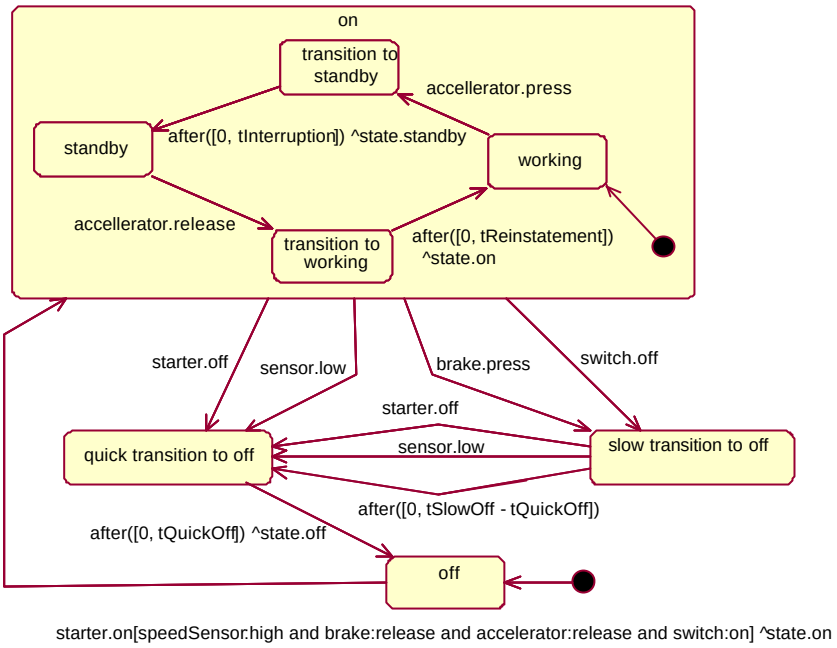

Figure 5: Enabler State Diagram.

In State Diagram of Figure 5 all the timing constraints for turning on, standby and off the regulator are reported.

Finally we have to specify the behavior of the velocity detector (SpeedSensor) –which sends an event when the velocity is higher or lower than the fixed speed of 50 Km/h, or 14 m/s)– and of the ControllerDisplay. They are both quite simple and self explaining (see Figure 6).

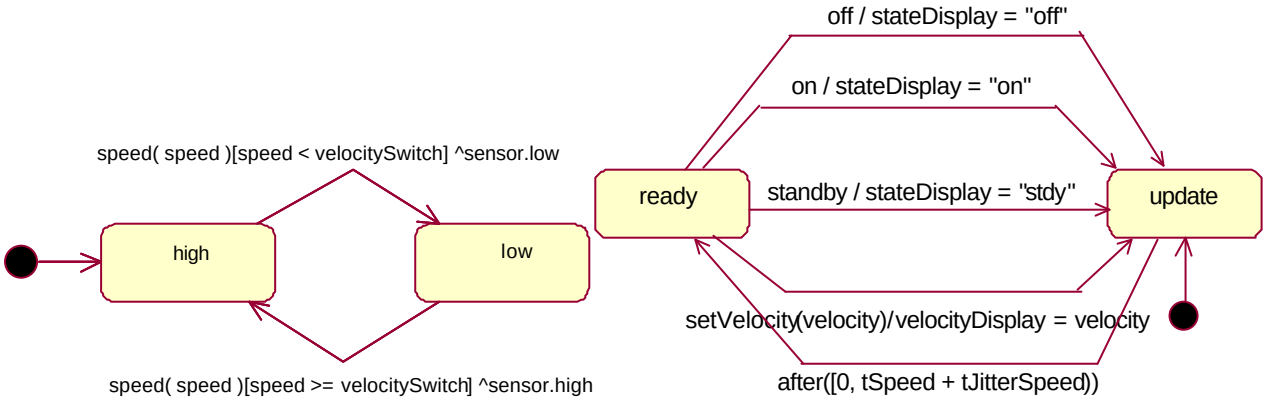

Figure 6: State Diagrams of classes Speed Sensor (left) and ControllerDisplay (right).

3. Constraints

In this section we discuss how the constraints of the case study are satisfied.

Response times of various instances are bounded by the following invariants:

```
speedFiletr.tFilter + speedFiletr.tJitterFilter <= clock.tClock - clock.tJitterClock
speedDisplay.tSpeed + speedDisplay.tJitterSpeed <= speedFilter.tFilter -
speedFilter.tJitterFilter
calculator.tCalculator + calculator.tJitterCalculator <= speedFilter.tFilter -
speedFilter.tJitterFilter
```

These constraints ensure that no single clock tick is missed by the speedFilter, and that speedDisplay and calculator do not loose any output of speedFilter.

The clock frequency is set by:

```
clock.tClock = 0.5s
```

The possible error is defined by `clock.tJitterClock`.

All the constraints of turning on, off and standby are operationally satisfied inside enabler:

```
enabler.tQuickOff = 0.1s
```

```
enabler.tInterruption = 0.2 s
enabler.tReinstatement = 0.25 s
enabler.tSlowOff = 0.5s
```

Only the velocity threshold of 50km/h (14 m/s) is modeled in speedSensor diagrams (speedSensor fires an event whenever the velocity threshold is exceeded):

```
speedSensor.velocitySwitch = 14 m/s.
```

Following the DESS methodology we would now translate the models described above into formal notations like TRIO [5] or Kronos [6]. This would allow us to show that the model satisfies the time requirements. For instance, the history checker of TRIO allows us to verify that stories (system evolutions, or scenarios) which represent acceptable behavior are compatible with the given model, while stories which represent unacceptable behavior are not compatible with the model. The Kronos model checker would allow to verify that the model satisfies given properties. In this particular case the most relevant property to be tested would be non-zenoness, i.e. the system may evolve indefinitely while complying with elementary constraints (like transition times).

However, this particular model is simple enough to allo2w analysts to assess the behavior of the system just by inspecting the model.

4. Conclusions

We have introduced a very simple extension of the UML language, taken from the work we are carrying out in the DESS project. Such extension, together with the usage of OCL, allowed us to specify the system proposed in [1], including the time constraints. It is relatively easy to show that the model described above behaves as required.

It should be noted that the DESS approach to analysis and specification would allow us to formally prove the properties of the system. This was not shown here because of space reasons, and because the case study was simple enough not to require a formal proof.

The DESS project also addresses the design, coding and testing phases. The contribution of DESS to these activities is reported elsewhere.

Acknowledgments

The work described here was partly supported by MURST funding of ITEA project DESS. Work by Vieri Del Bianco was partly supported by CiaoLab (http://www.ciaolab.com).

References

[1] *Case Study: a Car Speed Regulator*, http://www-leti.cea.fr/leti/UK/Pages/Tech_info/These/sivoes2001.htm.

[2] J. Rumbaugh, I. Jacobson, G. Booch. *The unified modelling language reference manual*. Addison-Wesley, 1999.

[3] OMG, *Unified Modeling Language Specification*. Version 1.3, March 2000. http://www.omg.org.

[4] Selic B., Gullekson G., Ward P.T., *Real-Time Object-Oriented Modeling*, Wiley, 1999.

[5] Ghezzi C., Mandrioli D., Morzenti A., TRIO, a logic language for executable specifications of real-time systems. *The Journal of Systems and Software*, 12, 2 (May 1990).

[6] Yovine S. Kronos: A verification tool for real-time systems. In *Springer International Journal of Software Tools for Technology Transfer*, Vol. 1, N. 1/2, October 1997.

[7] L. Lavazza, "An introduction to the DESS approach to the specification of real-time software", CEFRIEL report, April 2001.