Handling Timing Constraints with Virtual Timers (Position Paper)

Yvan Barbaix, Stefan Van Baelen and Karel De Vlaminck
Dept. of Computer Science, K.U.Leuven, Belgium
{yvan.barbaix, stefan.vanbaelen, karel.devlaminck}@cs.kuleuven.ac.be

The Position

The position we take in this paper is that the simple mechanism of virtual timers is powerful enough to annotate, analyze and verify (at run-time) the majority of the timing constraints.

Introduction

The subject of this position paper is the result of the work done in the resource constraint work package of the DESS project[1]. More concretely, this paper will present the project's way of tackling timing constraints, the most important non-functional constraint in many embedded systems.

It is important to stress that the partners in the consortium develop very diverse embedded products, ranging from cellular phones to avionics devices. Obviously, the approach we were looking for had to be usable in all these domains, using different programming languages and paradigms. Simple annotation of the design was not enough, Static formal proof of the compatibility of the constraints as well as run-time verification had to be feasible as well.

Basic Concept: Virtual timers

Our approach is based on the notion of *virtual timers*. Here is the fundamental reason why *virtual timers* are a good starting point:

> Thinking about timing constraints in a fine-grained way, we find that timing always requires two points of an execution trace; a start point and an end point. The start point is the point where an imaginary timer should start counting the time and the stop point is the point that we have to reach and where a given constraint should be valid.

So, we can think of a simple timing constraint as a constraint on some stop-watch that is started at some point and is halted at a second point -- the notion of a timer. In Figure 1 we show the two things we can do with a timer, resetting some time after its start and letting the timer time out. Note that we depicted this on a UML sequence diagram. As such the virtual timer is just an ordinary object.

[1] DESS is a European ITEA (Information Technology for European Advancement) project.

When a timer is started, we pass it a timeout time; the time after which we want the timer to time out. In situation (a) some object in the application resets the timer before it times out. This means that the time between the start point (x) and the end point (y) in the sequence takes less than *timeout* time units[2]. This defines the relationship '<':

$$executionTime < timeout$$

We will refer to this first usage of a timer as the start-reset sequence.

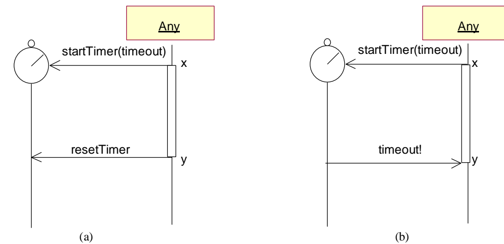

Figure 1: reset and timeout of a virtual timer

In situation (b) the timer times out at point (y). This means that the execution from point (x) to (y) in the sequence takes at least *timeout* time units. This defines the relationship '=':

$$executionTime = timeout$$

We will refer to this second usage of a timer as the start-timeout sequence.

Note that the relationship '=' is not defined as a stand-alone operator. Though this would theoretically be possible, it would make little sense to do so. It is impossible to indicate exactly at what point in the execution trace we would be at the exact time the virtual timer would time out. And even if it would theoretically be possible to express such a constraint, if would be impossible to verify it for a concrete implementation. One of the main guidelines in the DESS approach was that the constraints should not only be usable for theoretical analysis purposes, but should also be easy to map to run-time verification models.

Virtual timers are objects

Virtual timers are objects like all other objects in the application. This is important as it means that timers have an identity.

Virtual timers are virtual

[2] In most cases we will use *ms* or *s* as time units

Virtual timers are *virtual*. This means that they are conceptual things that do not exist in the running application. The net result is that they do not consume resources. Communication with a virtual timer like setting and resetting it is instantaneous. In addition, one can have as many virtual timers as needed to express the constraints.

No what-if relationship for timers

The two only interactions with a timer are the start-reset sequence and the start-timeout sequence. It is not possible to use a timer in a conditional expression of the kind 'if there is a timeout go this way, otherwise go that way'. This would mean that timers would no longer be virtual, but an actual part of the system.

Profiling UML

Given the definition of virtual timers, we believe that these virtual timers should be part of the standard UML definitions. In theory, we would prefer not to 'extend' the semantic meaning of UML -- the UML meta-model. And what we have seen so far could well be added to UML using just the standard extensibility mechanisms (stereotypes, tags and OCL constraints). However, we will also need extensions to UML that are not currently in the language. One such change could be replacing sequence diagrams with message sequence charts (MSC's). But no matter what way we would extend UML, in the rest of this paper we will assume that virtual timers are part of the UML variant in use.

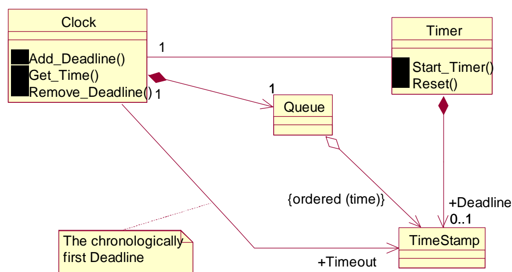

Figure 2: The relationship between Timers and the Clock

In Figure 2 we give an idea of how to look at virtual timers. Without going into details, we can think of the virtual timer objects as being linked to one clock. Every timer can have a deadline (timeout time -- a TimeStamp) associated with it. The clock keeps track of all timer deadlines via an ordered queue. The earliest timeout is the one the clock will signal when it would timeout. Whenever a timer is reset, its deadline is erased.

Mapping High-Level Constraints to Virtual Timers

Up until now, we only used simple timing constraints. And indeed, many constraints will be of the kind 'the time to execute this should not exceed *x* time', which can perfectly be expressed with the start-reset sequence of a timer. However, a majority of the timing constraints are more complicated that that. In these cases, we will have to provide a mapping from the timing constraints to the semantics of virtual timers. Here are some examples that indicate how such mappings are done.

Example 1

Let's take a look at the constraint 'The time to execute the code snippet should be 10ms with 1ms jitter'

Such a constraint actually indicates that there are two bounds for the execution time of the code snippet, an upper bound and a lower bound. The lower bound is then 9ms and the upper bound 11ms. As such, the answer to express the constraint of the execution should be to use two virtual timers like in Figure 3. Here, the indication is that the time between *x* and *y* is less than 11 ms and at least 9 ms. Note that the two timers now are two objects.

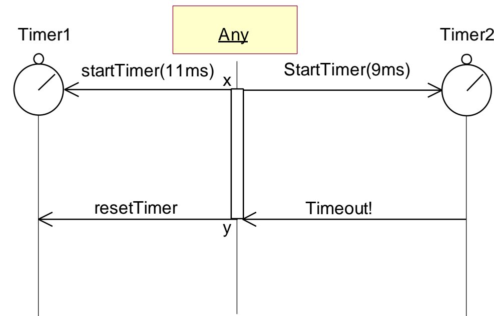

Figure 3: 10ms with 1ms jitter

Example 2

Here is a more complicated example: 'The period of the main cycle should be 50Hz'. Obviously, one could say that the solution is very simple using a timer with a timeout of 20ms. However, it is not that simple. A first question is, what does this constraint precisely refers to? Does it mean that every step in the cycle should be executed 20ms after its last execution or does it simply mean that for one point in the main cycle, the start of the execution should be 20ms after the previous one. In the last case, some jitter is allowed on the execution cycle of an inner loop point, but on average, it should be 20ms. Another question is, what is the jitter on the period? As we stated earlier, it is in general not possible to use the '=' operator in timing constraints, so an exact 20ms period is not possible. These questions can not be answered in a general way and will have to be defined per project.

In Figure 4 we give one possible solution. The solution takes the approach of allowing some jitter on the precise invocation time between two successive iterations. Over a number of executions however, the period will be accurate (there is no

error drift). Note that in this solution, the amount of jitter is not specified (although this could have been done). In the diagram, we have used some extensions to the basic UML sequence chart. The first one is the loop with a name *a*. Everything inside the dashed box is the body of the loop. The start-reset sequence makes sure that the execution of the code inside the loop does not take longer than one period. After completion of the loop body, we must wait some time before we can execute the next sequence. Because time flows from top to bottom, it is not possible to draw the following start-timeout sequence upward from the end of the loop body to the beginning of it. Therefore, we have introduced another UML extension, the loop continuation. It is the dashed box with specification [a: Loop-next]. This construct means that this is the next execution of the loop. The *a* indicates that it is the [a: Loop] that is being executed, so the sequence in this next loop step should be identical to the first one (but details can be omitted). Note here that the 2 *a*'s in the diagram specify the same spot in the code. The rectangle filled with horizontal lines indicates that this part of the code does not contain any code. We will cover more of UML extensions later in this paper.

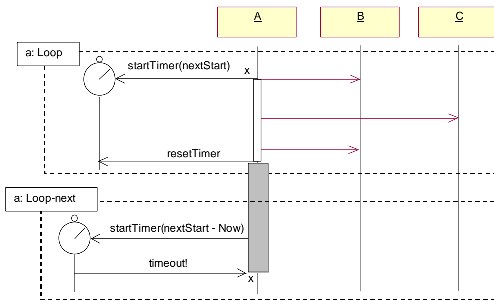

Figure 4: period constraint

Shorthand Notation

After these two examples, it should be clear that every more complex or higher-level constraint should be translated into some form of virtual timer expression. Even though the constraint type (like period in the second example) could be translated through many such mappings, we anticipate that the meaning of a constraint type will be the same for different constraints in the same application, or even across different applications inside a company division. As a result, we believe that such mappings from high-level constraint types to virtual timers should be grouped into UML packages that can be reused.

Using the explicit translation from a constraint type in a sequence diagram complicates the understanding of the diagram. This can be solved through the use of a shorthand notation for the constraint type. For example, the constraint in example 2 could be expressed as in Figure 5. A good CASE-tool should then be able to switch back and forth between the shorthand view of the diagram and the explicit one.

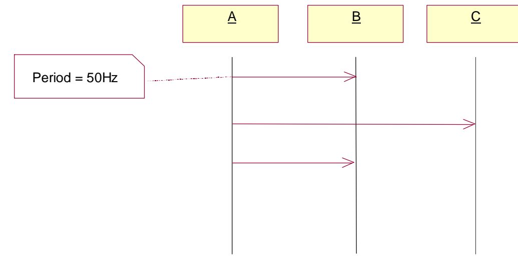

Figure 5: Short notation of the constraint

Constraint verification

Of course, annotating a design with timing info is nice, but if that is all there is, why bother? Things get interesting once we are able to check the consistency of our solution with the constraints. Fundamentally there are two ways of doing this, statically and dynamically.

Static Analysis

Static checking allows us to do the analysis before running the (final) application and, if all things are OK, to prove that the constraints will be met. On the downside this requires information about the whole system. This makes it much harder to do and puts some severe constraints on the whole system. For example, the exact behavior of the scheduler needs to be known, including context switch times.

To allow this sort of static analysis, we can not just put some constraint annotations in our system. We need to provide an analyzer with all possible paths through the system, which means that we have to provide all sequence diagrams of the system. This is currently not feasible with the semantics of UML. But as OMG is working on UML2.0, we believe things will improve in the near future.

Still, the basic notions of virtual timers are simple but powerful enough to allow static analysis methods.

Run-time Verification

The second way of checking constraint compliance is through dynamic run-time verification. This method can not provide a compliance proof, but only a high degree of confidence in the solution. On the positive side, run-time verification is relatively easy to perform.

The timers we discussed until now are virtual timers. Communication with timers does not consume any time nor do timers require any system resources. This makes them ideal for annotation and analysis purposes.

However, if we want or need to do run-time verification of timing constraints, we have to map the timing constraints into run-time timing mechanisms.

Concrete run-time objects

The most obvious way to do such a mapping is to translate the virtual timers into concrete timer objects that live at run-time. On the downside, we have to realize that some properties of the virtual timers can only be approximated by concrete timer objects. For example, communication with timer objects does consume some time. The number of timers allowed in the running system should also be limited.

While concrete timers are the most obvious way of handing our virtual timer constraints, other solutions are possible too. However, a discussion on this is outside the scope of this position paper.

Tool support

To support run-time verification of the constraints, tools should be extended to support automatic code generation. Although concrete code generators do not yet exist, we can give some issues generators should deal with. One such issue is that the developer should be able to activate and deactivate the verification of individual constraints. An application can have so many constraints that verifying them all could put such a weight on the system that the reason for constraint violations would be the load of the verification mechanism itself. Toggling individual verifications on and off would allow to deal with such items. Another issue is that the timers normally need to be passed as a hidden parameter to functions. As timers are objects, resetting a timer needs to act on the right object[3]. Passing these hidden parameters changes the (source) code. Good tools should be able to verify when such a parameter would not be needed and do some optimization without changing the interface. Finally, it is interesting to note that a start-timeout sequence will probably be mapped to a potential system call to a sleep function. This would then enable the OS the opportunity to schedule some other process.

UML, Sequence Diagrams and State Charts

Obviously, the project uses UML as its standard notation. However, we believe that some issues of the current standard are too weak for our needs. The weakest spots we found are mainly on sequence diagrams. Here are some problems we encountered:

- Sequence charts should have areas where the implicit top-to-bottom time order is 'switched off'. This would have a great potential of reducing the number of sequence charts needed to express some constraint fully.
- Sequence charts should have explicit annotations to indicate that all communications between objects are shown in the diagram. In theory, every sequence diagram is just a snapshot of an execution sequence where a number of details are omitted for clarity reasons. However, for our kind of applications, we often want to make sure that all details are present.
- Sequence charts should have better support for conditionals and loops and for composing and decomposing sequence charts

In general, we believe that looking more to the work done in MSC's would be a great first step in this direction.

Besides looking at sequence charts, we are working on the interplay between sequence charts and state charts.

Various Other Aspects

It is a fact that we can not express all possible timing or timing related constraints with virtual timer expressions. However, the DESS consortium is convinced that it is better to have a solutions for many but not all problems than to have no solution at all.[4]

One issue we did not speak about is the constraint violation policy of an application. Detecting a timing constraint violation is something that is tightly coupled with run-time verification of the timing constraints. If we can statically proof that the constraints will be met, there is little point in verifying these constraints at run-time. As a consequence, there is no need to provide a timing verification mechanisms that could detect constraint violations. However, once we do run-time verification, we have to decide what to do when a violation is detected. In our DESS project, we have build some ideas about handling such violations based on the conceptual ideas of design-by-contract. The case tool could provide a lot of support for such a policy. The important message here is that you have to know what your violation policy will be before rushing to code. The policy for the constraint violation handling could become part of the application in which case the design will be greatly affected

Conclusion

The approach for tackling timing constraints through virtual timers is rather simple. Nonetheless, we believe it has all the power to be usable and solve most of our needs.

---

[3] We will not go into more details here, but concurrency is the main reason why a hidden parameter is necessary.

[4] Timers are not the only things we use inside DESS, but the 4 page limit …