Generic Trigger Variables and Event Flow Wrappers in Reflex

Karsten Walther, Reinhard Hemmerling, and Jörg Nolte

Brandenburgische Technische Universität Cottbus
{kwalther,rhemmerl,jon}@informatik.tu-cottbus.de

Abstract. Reflex is a generic event driven OS for embedded devices. Event handlers and control functions are all represented by passive objects that are scheduled preemptively according to an earliest deadline first (EDF) strategy. The synchronization of events is based on an event flow model similar to the data flow paradigm. In this paper we will present the design rationale of Reflex and particularly discuss the language level aspects of our approach.

1 Introduction

Most small devices are more or less control loops for some processes in the real world. Therefore the operating systems on that devices do not need to be general purpose systems. Other requirements rule that world, like small memory footprint, robustness, real-time capabilities and of course resource sparing [4, 5]. Conceptually, most control systems are modelled around the notions of sen-

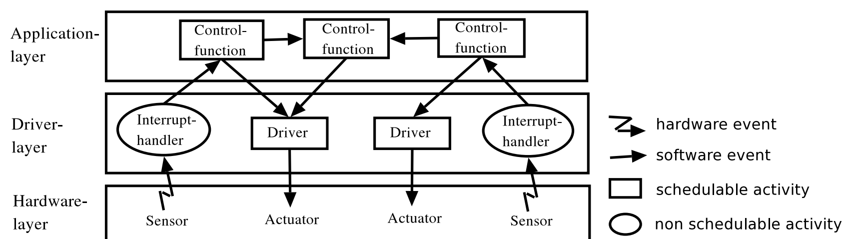

Fig. 1. An event driven control system

sors, control functions and actuators. Sensors measure aspects of the physical world and can trigger events that cause control functions to execute. The latter might in turn trigger actuators or cause other, more complex control functions to be executed [2, 7]. The interaction between these components can therefore easily be described using an event driven model. Some events are based on interrupts caused by the hardware while others are software based, causing other

parts of the system to act when specific things happened (fig. 1). REFLEX, the **R**eal time **E**vent **FL**ow **EX**ecutive therefore implements event processing and synchronization based on an event flow model described in section 2.

Since processes in the real world are controlled, most of these activities must be carried out within a certain, often hard deadline [1]. The whole control loop must therefore be decomposed into schedulable units that can be preemptively controlled by a scheduler. These units can either be represented by blocking threads waiting for events or simply by passive objects, that encapsulate e.g. a control function. Such functions are mostly simple and do not require an individual runtime stack. All state which needs to be preserved across multiple invocations of a control function can easily be kept in the corresponding object itself. This is a clear advantage over a thread based implementation, because embedded systems usually have to perform their job with severely limited resources and comparatively costly thread switches and multiple stacks can be avoided.

2 The Event Flow Model

All sensors, control functions and actuators are represented by objects that can communicate with each other by means of events. All respective classes are therefore rooted in an abstract `Event` class that provides only a pure virtual `trigger()` method to indicate that something important has happened. Classes can now handle events directly by simply defining the `trigger()` method.

Event handling is organized following a two level handling scheme. On the lowest level hardware events (interrupts) are directly handeled according to the hardware priority of the respective devices to be controlled. On the second level software initiated events are handeled by event handler objects (activities in our terminology) that are scheduled according to an Earliest Deadline First (EDF) strategy. The latter usually perform the postprocessing of interrupts as well as complex control tasks (fig.1).

Device driver objects are therefore located on the lowest level and handle hardware initiated events directly on the call to `trigger()`. The latter is issued by a low-level assembly part that bridges true hardware events to C++-objects representing devices and their drivers. These handlers might trigger other activities, that are now scheduled according to their deadline. All classes that handle schedulable events are derived from an `Activity` class that defines a pure virtual `run()` method and puts the object on the list of the scheduler when the object is triggered.

Thus an event flow can already be established that propagates a single hardware event through a user defined chain of objects reacting on that event within individual deadlines. The main advantage of this scheme is the implicit activation of objects in case their inputs become valid. However, often it is desirable to propagate a conglomeration of values along with an event to ease communication and synchronization between the objects. Synchronization and scheduling in REFLEX is therefore based on an event flow model that is in principle very similar to a data flow model. Control functions are represented by passive C++-objects

whose control methods will be automatically scheduled for execution when all of their input values are defined. Such schemes are implemented in REFLEX with generic trigger variables and event flow wrappers.

```
class TemperatureMonitor: public Activity {
      TriggerVariable<float> current;
      float previous;
public:
      TemperatureMonitor (TimeUnit deadline)
          : Activity(deadline), // initialize Activity with deadline
          current(this) {}  // associate variable with "this"

      // store value and trigger the excution of "run()" atomically
      void tempChange(float value) { current = value; }

      // "run()" is EDF scheduled following write to "current"
      void run()
      {
          float curr = current;  // atomic read!
          float diff = curr-previous;

          if(diff > THRESHHOLD)   // a discontinous increase
              cout << "Alarm: " << curr << "diff " << diff << endl;

          previous = curr;
      }
};
```

Fig. 2. Example use of a trigger variable

2.1 Trigger Variables and Event Counters

The most simple trigger variable can store a single value of a specified type. When a value is written to that variable the value is stored and an associated object is automatically triggered. When that object is a schedulable `Activity`, the `run()` method will now be scheduled according to the specified deadline (fig. 2). When `run()` is eventually executed, it can access the trigger variable and read the stored value without further explicit synchronization. Thus trigger variables represent a single buffer that can be read or written atomically while the triggering mechanism enables an effective means for data flow synchronization. We provide several types of trigger variables with slightly varying semantics. All ensure that the associated object is triggered at most once before the variable is read. Certain types of trigger variables might allow multiple writes to the single buffer to provide the most recent value while others allow only a single write.

Note, that the `run()` method is never overlapped with itself on the same object.

```
class WatchDog: public Activity {
   TriggerVar<float> pressure, temperature;
   EventCounter counter;
public:
   WatchDog(TimeUnit deadline) : Activity(deadline),
       counter(2, this), // trigger "this" when 2 events happend
       pressure(counter), // associate pressure with "counter"
       temperature(counter){} // associate temperature with "counter"

   void tempChange(float value) { temperature = value; }
   void presChange(float value) { pressure = value; }

   //  sched. "run()" when both "temperature" and "pressure" are defined
   void run() { . . . }
};
```

Fig. 3. Indirect triggering of an Activity

When more events are generated than the receiving object can handle all additional values associated with the events are lost. Although this should rarely happen in a well designed real time system, we provide `Fifo` objects that can hold up to N values instead of just one to alleviate this problem. Any time a `Fifo` is read the associated `Activity` object is triggered again when the fifo is not empty. Thus the processing of each fifo element is scheduled in an EDF fashion.

Trigger variables can be combined with event counters to implement an event driven data flow scheme with multiple parameters. Each parameter is represented by a single trigger variable. These variables do not trigger the `Activity` directly but indirectly through an `EventCounter` instance (fig. 3). Thus the `WatchDog` shown in figure 3 is only scheduled, when both temperature and pressure are defined. Note, that `Fifo` objects can be applied in this scheme, too.

2.2 Event Flow Wrappers

The scheme discussed in the previous section can easily be generalized by means of generic event flow wrappers. In figure 4 we describe such a wrapper for binary functions. Similar template classes can easily be created for functions with any numbers of parameters. Now it is possible to construct arbitrary data flow graphs from existing functions or function objects, where each function is represented by a node in this graph. Synchronization is implicit and each function is scheduled according to its deadline.

```
template<typename FUNC, typename ARG0, typename ARG1, typename RESULT>
class BinOp: public Activity {
   EventCounter counter;
   FUNC func;
public:
   TriggerVar<ARG0> in0; // input 0
   TriggerVar<ARG1> in1; // input 1
   TriggerVar<RESULT> &result; // reference to output

   BinOp(TimeUnit deadline, RESULT& output) : Activity(deadline),
       counter(2, this), // trigger "this" when 2 events happend
       arg0(counter), arg1(counter), // associate args with "counter"
       result(output) { } // associate "result" with "output"

   // "run()" is EDF scheduled when all inputs are defined
   void run() { result = func(in0, in1); }
};
```

Fig. 4. A generic wrapper for binary functions

The simple scenario in Figure 5 sketches how such wrappers can be applied. The outputs of `sensor0` and `sensor1` are connected to the inputs of a `BinOP` Instance that calculates the difference between both sensor values and provides the input for a `Display` function that displays the difference. When the sensors now periodically measure the temperature the computation of the difference of both sensors is automatically scheduled once the values are available. The same holds for the display procedure that is automatically scheduled when its input is defined.

```
Display<float> display(deadline, "Temperature Diff: ");
BinOp<minus, float, float, float> diff(deadline, display.in0);
Temperature sensor0(diff.in0);
Temperature sensor1(diff.in1);
```

Fig. 5. Use of generic wrappers

3 Related Work

Several operating systems for deeply embedded devices exist today. One of the most well known is probably TinyOS [6]. TinyOS is an event driven system implemented with the NesC-language, a C dialect for deeply embedded systems

[5]. TinyOS has a truly small memory footprint but does neither support preemption nor event driven data flow synchronization like Reflex. Furthermore, Reflex is entirely based on standard language features of C++ and does not rely on a specifically designed programming language. The memory footprint is therefore higher as in the case of TinyOS but it is still acceptable (in the order of a few KB code) for most applications.

Our interrupt handling scheme is strongly inspired by the design of Pure [3] as well as its predecessor Peace. Pure focusses on highly tailorable operating systems for deeply embedded devices and has clearly proven that C++ can effectively be applied in the context of embedded systems. Compared to Pure the Reflex platform strongly focusses on an event driven programming model with data flow synchronization.

4 Conclusion

In this paper we have shown how generic and object oriented programmming paradigms can be applied to implement event flow schemes for embedded systems. The concept of trigger variables significantly eases synchronization for simple control tasks. Generic event flow wrappers effectively support the composition of control loops for embedded devices. Existing components can now be embedded into an event driven data flow synchronization scheme with deadline scheduling without being aware of this fact.

References

1. I. Bate and A. Burns. An integrated approach to scheduling in safety-critical embedded control systems. *Real-Time Syst.*, 25(1):5–37, 2003.
2. S. Bergbreiter and K.S.J. Pister. Cotsbots: An off-the-shelf platform for distributed robotics. In *In Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems*, october 2003.
3. Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schroder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. On the development of object-oriented operating systems for deeply embedded systems - the PURE project. In *ECOOP Workshops*, page 26, 1999.
4. Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.
5. David Gay, Philip Levis, and Robert von Behren. The nesc language: A holistic approach to networked embedded systems. In *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design an Implementation*, June 2003.
6. Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In *the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, MA, November 12–15 2000.
7. Klas Nilsson, Anders Blomdell, and Olof Laurin. Open embedded control. *Real-Time Syst.*, 14(3):325–343, 1998.