Adapting Futures: Scalability for Real-World Computing

Johannes Helander
Microsoft Research Redmond
jvh@microsoft.com

Risto Serg
Tallinn University of Technology
risto.serg@dcc.ttu.ee

Margus Veanes
Microsoft Research Redmond
margus@microsoft.com

Pritam Roy
University of California, Santa Cruz
pritam@soe.ucsc.edu

Abstract

Creating robust real-time embedded software is critical in combining the physical world with computing, such as in consumer electronics or robotics. One challenge is the complexity of dealing with time together with implementation details that often end up implicitly determining the temporal behavior of the program.

In this paper we suggest deconstructing a program into two separate aspects, the functional implementation and a temporal pattern, each expressed separately in a different language. This separation enables independent specification, analysis and prediction of the temporal behavior without regard to the implementation. Meanwhile the implementation is optimized for platforms with different capabilities through a scalable programming model that automatically adapts the execution to the level of concurrency a platform can support. Finally the aspects are put together to create a working system.

This paper presents the use of futures and partitures to achieve predictability and performance in embedded systems. A new real-time scheduler for partiture based futures execution is introduced, along with multiple implementations, including one for an 8-bit microcontroller. The paper explains how to create, model, and execute futures and partitures.

1. Introduction

Computing devices that interact with the physical world and with other computers are at the core of new consumer applications, such as games, appliances, medical devices, and communications gadgets. Applications of computing combined with the real physical world also include robotics, industry, and civic infrastructure. Multiple devices are often needed to achieve a single goal. They must interoperate and adapt to an often chaotic environment.

Scheduling activities on a computer for real-time and energy efficiency requires maximal information about the temporal behavior of a program. The information on timing of different phases of a program, as well as dependencies between different parts of a program are traditionally implicitly encoded into the implementation of the program, mixed in with other program logic and related details. Static analysis can sometimes extract some of the information, but reverse engineering the programmer intent from an implementation is an unnecessary indirection. It would be better if the programmer simply would declare her intent directly.

In this paper we propose using two programming paradigms new to embedded systems design to let the programmer easily express the intent of sequential dependencies and temporal parameters.

1) The ***future***, originally proposed in 1976 [14] and was the principal synchronization construct in MultiLisp [15]. It is used to express concurrency and potentially parallelism or sequentiality between program components. By letting the programmer freely and lightly express parallelism in the program it is possible for the runtime system to adjust the level of parallelism to the level that is optimal for the target system. This means increased parallelism on a multi-core or multiprocessor, where it is advantageous to keep the cores as busy as possible with the aim of maximizing the gaps between executions where the processor can be put into a power saving mode. On the other hand it means decreased parallelism, even completely sequential execution on a small microcontroller that does not have room for multiple execution stacks—often the biggest single RAM footprint cost.

The future allows a common programming paradigm for cyber-physical programs across virtually the entire spectrum of programmable devices. We will examine in further detail in section 2 how futures are created, how they are executed, and how they and the program code can be optimized to different size computing devices.

2) The ***partiture***, analogous to the score of a symphony orchestra, is a separate program that describes the temporal aspect of an application, leaving

implementation details to the regular programming language, such as C or Java/C#. The partiture is a collection of scores, where each score describes the expected sequence of events and message pattern of a single task as a collection of bars. Each bar prescribes the estimated time and duration when something should happen, as well as where messages are sent and what other bars are triggered. The partiture does not prescribe what the content of messages is, what functions are called, or what the implementation does. The analogue is the conductor of an orchestra who may be most interested in when and at what tempo instruments play but not so much what the actual notes are.

The partiture is an estimate of the temporal behavior. It may be authored by hand, generated from an execution log, or generated by analysis tools, or as a combination of the above. If the partiture is based on solid worst-case analysis, it is a hard real-time partiture. If it is based on statistical analysis, as shown in section 6, it is a stochastic real-time partiture. The partiture can be updated and adapted during run-time if so desired [2][3], or hard-coded for the life of the device. Examples of both cases are shown in the results section.

1.1 Modeling and adapting partitures

A partiture consists of a number of scores, each describing the temporal behavior of one task. Bars within a score are usually related and form a dependency graph, where one bar triggers another. Separate scores meanwhile are unrelated but a process represented by one score may send messages to processes represented by other scores. In the current implementation the partiture is expressed in XML syntax but could obviously be expressed in a different syntax as well. The partiture could also be written into e.g. C# attributes and separated and collected using the regular C# metadata tools into a separate file.

```
<partiture>
  <score name="simple-sample">
    <bar name="dosome" duration="PT0.002S"/>
  </score>
</partiture>
```

Figure 1.1: A very simple score in a partiture stating that dosome will take two milliseconds to complete.

A partiture can be viewed as either a program or as data. When viewed as data it is natural that the partitures can be sent around in messages, combined into a database, converted to model programs, etc. The ability to combine multiple partitures with each other and data from traces with existing partitures, makes it relatively easy to analyze and adapt the partitures. An example of the description can be seen in figure 1.1. Once the partiture has been adapted, it can be sent back to the scheduler of the worker machine, where it is executed by the scheduler, as will be seen later in this paper.

1.2 Scheduling futures

Futures are encapsulated method calls that are evaluated (executed) at some point in time from when they were created until when the result is used. The difference to a thread is that 1) it is assumed that futures are cheap to create and terminate; 2) the method called can take any arguments; 3) the call site of a future looks roughly like a regular method call; 4) there is no inherent guarantee that the futures will in fact run concurrently and fairly. Another related term is a delegate (an anonymous function call); the difference is that the future is evaluated at any time at the discretion of the scheduler. Creating delegates usually also requires special syntax and support in a programming language.

```
ISimpleService *proxy;
status = Factory->v->Create(Factory,
         actual_object, "ISimpleService",
         "simplescore/dosome", completion,
         &proxy);
status = proxy->v->print(proxy, "test");
DoOtherWork();
status = completion->v->Wait(completion);
```

Figure 1.2. Invoking a method as a future through a proxy object. The Wait on the optional completion object returns after the future has completed.

A future can be created in a number of ways: 1) from a local method call, where the object has been replaced with an interposition agent object that turns a regular method call into a delegate and hands it off to the scheduler; 2) from a local call into a futures factory (part of runtime) with the stack frame replaced by a structure; 3) from a pre-initialized structure; 4) deserialized from a message that matches the method call; or 5) from a string or file that contains the serialized representation of the call. In all cases some metadata is required: (1-3) can get by with the size of the stack frame, the object the method is applied to, and the correct method. Cases (4-5) instead require complete metadata for serialization and deserialization of all the parameters. In our implementation the methods of any abstract class, native or managed, can be turned into a future, given that it is a subclass of one of the known base classes.

A future does not require a stack until it is ready to execute, thus to save memory the stack allocation can

be deferred until the actual execution. This means that stacks can be efficiently recycled thus significantly decreasing the cost of creating a future and the RAM footprint of an application.

Since futures can be executed when the scheduler decides to do so they can be executed in a serial fashion one at a time, providing low memory utilization, or many at once, providing high CPU utilization. Since futures are cheap to create, it is expected that programmers will not hesitate to use them to express inherent concurrency present in their application.

1.3 Putting them together

When and in what order futures are executed is part of the temporal behavior of the program and is expressed in the partiture. The partiture also contains information on dependencies between the bars. A dependency will determine a partial ordering between bars and consequently the execution order of the futures that are matched with the bars. If there is a cycle in the dependency graph then the futures cannot be run serially, since there is no partial ordering; consequently multiple stacks and pre-emptive scheduling is required—unless the cycle indicates a deadlock in which case there is no valid schedule at all. The dependency graph is currently created manually but will eventually be produced by the analysis tools.
If the method that implements a future blocks during its execution, for instance waiting on a timeout or some other execution to complete, the future moves from one bar to another (or another incarnation of a repetitive bar). The blocking operation or the blocking object needs to be tagged to demarcate the correct bar to switch to.

The scheduling model and algorithm are further explained in section 4.

1.4 Scenarios

Let us consider three sample systems that serve as examples later in the paper. #1, simpleservice, implements a simple object that can print a string; #2 hardsample, is a basic sensor application that does periodic sampling of a microphone, some filtering and correlation of the data, and finally outputs a result on a serial line. It also listens to commands, such as reset, from a host computer on the same serial line; #3 is a distributed system that does sensor reading and then sends the result to another device over a wireless network with variable performance and thus the need for adaptation.

The rest of this paper is organized as follows: Section 2 explains how to turn an application into a scalable program that uses futures and partitures. Sections 3 and 4 detail the execution and scheduling of partitures and shows how the system can be optimized for extremely small embedded microcontroller applications (measurements included). Section 5 introduces an abstract model of the scheduling that can be used for robust systems development (results included). Section 6 shows how the partiture can be used for distributed scheduling (measurements included). Finally section 7 looks at related work.

2. Programming with futures

On the client side (the caller) a future looks like a regular method call. The future can represent either a remote call, a call to another runtime (e.g. C to C#), or a local call. The caller may wait for the call to complete at a later time as required. The programmer needs to be aware that any return values (out argument) will become valid only after the call has completed. The client side can create multiple futures and then wait for them to complete. If the method calls in the futures are not expected to return values there is no need to wait for them.

On the server side (the callee), the future looks like a regular method call. When the call returns it is considered complete and any waiters will be woken up. When a future is created (such as in figure 1.2 above), it should be associated with a bar in the partiture. The shared label lets the scheduler match the future with the correct temporal description and use the right temporal parameters for scheduling its execution.

```
for (;i < 100;) {
    wait();
    obj->read(obj, 0x1000200);
    obj->filter(obj);
    obj->output(obj, "com2");
}
```

Figure 2.1: A simple sensor application control loop.

A common pattern for a sensor processing application is to wait until it is time to sample a sensor, read a value from hardware, apply a digital filter, then send the result somewhere, e.g. over a serial line (figure 2.1). The same pattern in general is common in almost any program, in a loop wait for something, process it, and then wait some more. The loop construct is inefficient as it forces a separate thread to be created for each such task, meaning a separate stack, meaning a lot of memory. Instead with the futures the loop is removed altogether and instead replaced by a partiture with a score that has the loop encoded as a temporal pattern. The application just creates the futures, providing a method call template (figure 2.2). The futures are associated with the right score by

naming the score while creating the future. The method name is here used to match the label in the score rather than specifying the label explicitly. The wildcard matches the same future with all incarnations of a bar. The score contains the dependencies between the bars (the triggers) as well as the estimated execution times of each bar and the loop count. In figure 2.3 the sending of the response is in addition split into multiple chunks so as to process one byte at a time.

```
future=create(...,"hardsample/*/*");
future->read(future, 0x1000200);
future->filter(future);
future->output(future, "com2");
```

Figure 2.2: Creating futures for the sensor example.

The conversion of the loop into a partiture allows more efficient memory use since a stack is not needed between the calls to the methods as the implementation code of the loop has been removed. The score also contains much more precise timing information allowing the schedule to be analyzed and massaged to a given platform and circumstances. The server side (the implementation of the methods) stays exactly the same and does not need to be revised for the purpose of becoming a future. However, for further performance enhancement purposes it may later be desirable to revise the code, such as the byte sending code sending just one byte at a time rather than internally waiting for the serial line to drain. This allows the stack to be taken away also during the sending phase of the program.

```
<score name="hardsample">
  <bar name="read" duration="PT0.001S"
       slack="PT0.0005S">
    <repeats count="1000" offset="PT0.02S"/>
    <trigger name="filter" />
  </bar>

  <bar name="filter" duration="PT0.001S">
    <trigger name="output" mode="1/100"/>
  </bar>

  <bar name="output" duration="PT0.0002S"
       slack="PT0.0001S">
    <repeats count="4" offset="PT0.001S"/>
  </bar>
</score>
```

Figure 2.3: A score that replaces the control loop.

One important concept is in fact the incremental nature of developing code for various platforms, allowed by the futures. Instead of having to immediately completely redesign an application to run on a small device and then again having to redesign it for a more powerful platform in the second revision of a product, such as would happen with so-called split-phase operation [22], the same program can be run on multiple platforms and be optimized one step at a time.

It would be feasible, and practical, for instance, to take a legacy multithreaded program that has a few threads, each doing loops with blocking operation, and turn them into futures. Since the native futures only work for abstract classes the program would first have to be written in an object-oriented fashion. The program could initially be run on a larger platform that can handle multiple execution stacks. To adapt the system to run on fewer execution stacks to save memory, the blocking points would be converted to futures one at a time, all the while the application would continue to work. In the extreme case all blocking points could be converted into futures, meaning the application could run on a single stack, shared perhaps with other applications, without any pre-emption or even interrupts.

The same application would still continue to run on more powerful computers, however. In fact given a multicore computer, the scheduler would *automatically* be able to run the futures in parallel, thus increasing the utilization of the processors. Since the score contains dependency information the parallelism would be scaled to the extent there is inherent parallelism in the program and hardware resources are available. Surprisingly the optimizations done for the 8-bit microcontrollers are useful also for the high-performance multicore processors. This conclusion is in line with prior use of futures in parallel languages, where the purpose is specifically to increase parallelism; as well as with preliminary experiences with our alternate implementation of futures on threads, when run on multiprocessors.

3. The execution of futures

To execute futures and partitures, some system support is required. We split the scheduling into two sub-problems: 1) the execution of futures; and 2) the interpretation of partitures, and then make the two work together.

The futures can be executed on top of threads, where one thread executes one future at a time; or they can be implemented natively directly by the scheduler without the need of threads. A future can be viewed as progressing through temporal phases, with interaction points delimiting the phases. The interaction points are where the future blocks on a resource, waits for a timeout, or signals a resource potentially unblocking another future. A simple future has just one phase, and runs to completion. If all futures are simple single-phase futures then no pre-emption is required. The execution state machine for a single-phase future is

depicted in Figure 3.1. In this case the state machine of the future execution matches that of the corresponding bar in the score. The pre-wait state is where the future is ready to execute but it has not yet been assigned a stack. To move from the pre-wait to the ready or run state a stack is allocated and the frame copied from the template in the future to the actual execution stack.

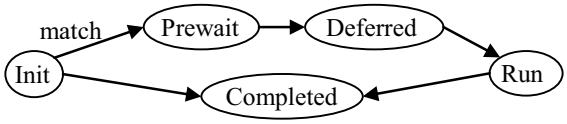

Figure 3.1: Future execution state machine with a single temporal phase (run-to-completion semantics).

When a future looks more like a thread and in fact blocks and otherwise interacts at multiple points with other futures, the state machine is more complicated. Each temporal phase corresponds to a bar—thus the bar keeps changing during the execution and the state machines are more loosely coupled. Figure 3.2 depicts the state machine for a multi-phase future. Rematch transitions the future from one bar to another.

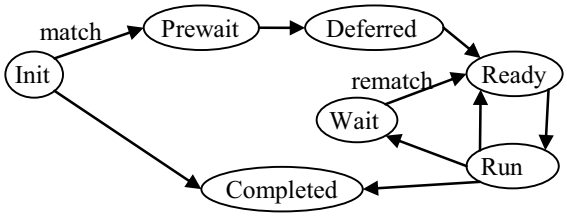

Figure 3.2: Future execution state machine with multiple temporal phases.

Once a future has been given a stack it cannot relinquish the stack until it has completed. Thus pre-emption is intrinsically required for scheduling multi-phased futures. If the dependency graph in the score, as defined by who triggers whom, has cycles then full pre-emption and multithreaded execution including multiple stacks is required for the program to be runnable—assuming any cycle does not indicate an outright deadlock. If the dependency graph is acyclic then a half-pre-emptive scheduler will suffice. The following four execution modes are possible:

1) Run-to-completion: all futures are single-phased.

2) Half-pre-emptive: the dependency graph is acyclic.

3) Pre-emptive single-core: there are no temporal overlaps.

4) Multicore: the dependency graph does not fully dictate the execution order but parallelism is available.

The execution modes possible can be deduced from the score by means of graph analysis. Section 5 shows how the dependency and scheduling graph can be created and analyzed from the score using the nModel modeling framework.

4. Scheduling the score

The scheduler is essentially an interpreter of the partiture, a domain-specific reactive programming language. A bar contains the information of the timing as well as triggers at completion. The scheduler acts on bars and drives them through the state machine in figure 4.1. In order to run a bar must be triggered, have an associated (matched) future, and the current time be between the earliest and the latest start times. If the bar is not ready by latest start time, it will move to the expired state. The score may contain a predefined alternate future (delegate) that is executed as a result of expiration, providing a timeout exception handler and causes the bar to transition to the executing state. Of all the bars that are in the run state the scheduler will assign hardware resources to the bars as appropriate. In a hard real-time scenario there can never be any contention—the score has been analyzed ahead of time. In a soft real-time scenario the score is adapted at run-time and is executed on a best effort basis.

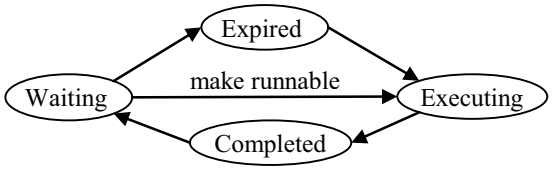

Figure 4.1: The state machine of a single bar.

When futures are implemented on top of threads, the low-level thread scheduler handles the execution machinery, leaving only the partiture interpretation and admission test to be implemented by the messaging system or other middleware. Results from applying this technique to distributed web services messaging are showed in section 6. When the thread scheduler works on multiple processors, the schedule becomes automatically multicore. A native multicore implementation has not yet been attempted at the time of the writing of this paper.

Perhaps more interestingly the futures scheduling can natively be implemented in a very small footprint. In a hard real-time scenario, such as the score "hardsample" presented earlier in this paper, the schedule needs to be completely analyzed ahead of execution to ensure that execution is possible and that the estimated durations are correct worst case values. If scheduling is possible in a run-to-completion fashion,

as determined by the model analysis (see the following section) there will be one or more scheduling sequences that are a valid implementation of the partiture. Picking one of the valid sequences yields a sequence of instantiated bars and times.

In the hard real-time example there is typically very little dynamicity in any single application. Thus the futures that are to be executed are in fact known ahead of time. This means that they do not need to be created at run-time from either e.g. C code or XML messages [1] but can be declared ahead of time. For this purpose we have added an extension to the partiture language that allows inserting the expected message directly into the score, just as if the message would have arrived during run-time. Now instead of deserializing the message at run-time into a future and then matching the future against the correct bars in the score, the deserialization and matching is done at compile time. An example is shown in figure 4.2.

```
<score name="hardsample">
  <bar name="read" …>
    <future>
      <wsa:To>object:sensor</wsa:To>
      <sensor:read>
        <sensor:memoryaddr>0x1000200
        </sensor:memoryaddr>
       </sensor>read>
    </future>
  </bar>
```

Figure 4.2: A predefined future within a bar in the partiture. The future is a method call to the sensor object with parameters included, i.e. sensor->read(sensor, 0x1000200).

Since almost everything is constant, most things can be put in read-only memory, usually much cheaper and more abundant in embedded systems. The notable exceptions are the execution stack and the sensor object itself. The sensor object is the implicit *this* argument to all method calls and is used to hold the states of the sensor reading, filtering, and output processes, in a typical object oriented fashion.

The actual score is a sequence of bars to be executed at given relative times. The scheduler simply keeps an index to the current bar. When it gets to the end it goes back to the beginning. The scheduling and execution proceeds as follows:

1) The scheduler picks the next bar.

2) The scheduler compares the earliest start time with the current time. If the time has not yet arrived the processor will be put to sleep mode until the latest start time.

3) The future to be executed is pointed to in the instantiated bar (pre-matched). The future contains the count of parameters to the method call.

4) The parameter count is compared to the maximum allowed for register arguments. Any overflow parameters are copied to the execution stack.

5) The remaining parameters are copied to the argument registers. The function pointer is copied to a temporary register.

6) A subroutine call is made to the location of the temporary register. The method call in the future is thus called with the correct arguments.

7) When the future completes it returns back to the scheduler, which repeats the loop from #1.

The control loop is very small. However, due to stack and register manipulations it will have to be written in assembly code, although steps 1-3 can be done in a C subroutine. Since the loop never returns the scheduler index counter can be kept in a callee saved register. There is zero bytes of RAM used by the scheduler that is not mandated by the machine calling convention. The implementation of the hard real-time futures scheduler uses 516 bytes of ROM on an AVR microcontroller. Figure 4.3 has the size breakdown of the sample application and OS support on an Atmel ATmega169P microcontroller. It was possible to find a fixed schedule for the sample application and consequently store the bars and futures into ROM. The code size of the scheduler is fixed and not application dependent. The memory required for futures, bars and the application itself, depends on the complexity of the application.

bytes	Scheduler	Futures	Bars	Application
RAM	0	0	0	145
ROM	516	25	28	3638

Figure 4.3: Size breakdown of sample application with scheduler fixed overhead, and application dependent overhead included. The scheduler does not use any RAM.

5. Modeling the partiture

A partiture may in general contain multiple scores. Each score is a set of bars that implies various causal relations between the bars that have to be maintained during execution. In this section we provide a behavioral model of a single score of bars, called a

score model. The score model can be used for runtime analysis and for static analysis of the scheduler.

As our modeling language we use *model programs* written in C# within the *NModel* framework [4] that is freely available for download [7]. The score model is divided into separate partial models that are composed into the complete score model using (parallel) *composition* [6] of model programs. The three partial models we are considering here are: *Triggering, Counting, Timing.*

Each of the partial models is given by an independent model program that describes a set of valid traces of bars or *schedules* that takes into account only part of the information in the bars. Thus one partial model may allow schedules that are not possible in another partial model, whereas their composition enables only schedules that are possible in *all* of them. In other words, the composition does not cause emergent behavior in form of traces that were not possible in any of the partial models, but preserves trace intersection.

5.1 Triggering

The triggering model specifies the aspect of a score that deals with one bar "triggering" another bar. For example, in the score in Figure 2.3, the bar named "read" triggers the bar named "filter". It is assumed that no two bars have the same name within a score.

In NModel, a model program is scoped and identified by a namespace and by default all the fields of all the classes in it constitute the state variables of that model program. In the remainder of this subsection all the definitions are assumed to be given within the scope of the Triggering namespace.

```
namespace Triggering { ... }
```

There is a main class Contract that has a state variable triggers that is assumed to be initialized from a given score (such as the one in Figure 2.3, that will be used as an example) with a map from bar names to a set of triggers. Each trigger has a bar name that is being triggered and mode information telling whether this bar is triggered always, only initially, or with a certain frequency (after each time the triggering bar has occurred a certain number of times). This initialization step is done through an Init action from a given score (the details of which are omitted here for brevity). Note that, a bar may, in general, trigger several other bars.

```
partial static class Contract { static Map<string, Set<Trigger>> triggers; }
public class Trigger : CompoundValue
{ public string name;
  public Mode mode;
  public int frequency;}
public enum Mode { Always, Initially, Frequently }
```

There is a state variable readyBars that includes the names of the bars that are ready to be executed by the scheduler. The value of readyBars is initialized by the Init action. We say that a bar is ready in a given state if readyBars contains it.

```
public static Set<string> readyBars;
```

In NModel each model program has a set of *actions*. An action is a term that may take arguments and is associated with a guarded update rule. The *guard* or the *enabling condition* determines if the action is enabled in the give state, and the *update rule* describes the state transition caused by executing that action.

The Triggering model program has an Execute action that takes the name of a bar as an argument. The action is enabled in a given state if the bar is ready. The guard is given by the Boolean function whose name starts with the name of the update rule method and ends with Enabled. (Guards are similar to preconditions in SpecExplorer [8].) The update rule removes the triggering bar from readyBars and, depending on the mode, adds the triggered ones to readyBars. It also updates the state variable occurrences, which keeps track of the number of times each bar has been executed so far. The number of occurrences of all bars is initialized to 0 by the Init action.

```
class Contract {
  static Map<string, int> occurrences;
  [Action]
  static void Execute(string b)  {
    readyBars = readyBars.Remove(b);
    foreach (Trigger t in triggers[b])
      if (t.mode == Mode.Always ||
          (t.mode == Mode.Initially && occurrences[t.name] == 0) ||
          (t.mode == Mode.Frequently &&
           occurrences[b] == t.frequency * (occurrences[t.name] + 1)))
        readyBars = readBars.Add(t.name);
    occurrences = occurrences.Override(b, occurrences[b] + 1); }
  static bool ExecuteEnabled(string b) { return readyBars.Contains(b); }}
```

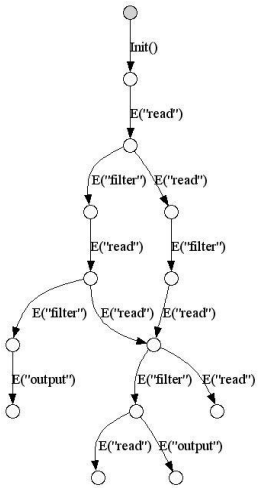

Figure 5.1: Triggering Model Scenario.

A partial exploration of the triggering model is illustrated in Figure 5.1 that has been created with the model program viewer utility of NModel. The action name Execute has been abbreviated by E. The state variables are initialized according to a score similar to the one in Figure 2.3, where it is assumed that the Init action sets readyBars to the set containing "read", "read" triggers itself and "filter". Here triggers maps "filter" to a set containing the trigger with name "output", mode Mode.Frequently, and, instead of 100, frequency has been initialized to 2. A path from the initial state shows a prefix of a schedule that is possible according to the triggering model.

5.2 Counting

The counting model has a single state variable barCounter that specifies the number of times that each bar in the score is to be executed. The initial value is set by the Init action for a given score. Each time a bar is executed, the count of that bar is decremented by one.

```
namespace Counting {
  static class Contract {
    static Map<string, int> barCounter;
    [Action]
    static void Execute(string b)
    {
      if (barCounter[b] == 1) barCounter = barCounter.RemoveKey(b);
      else barCounter = barCounter.Override(b, barCounter[b] - 1);
    }
    static bool ExecuteEnabled(string b)
    { return barCounter.ContainsKey(b); }
  }
}
```

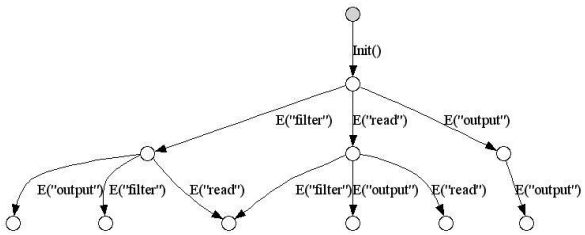

Figure 5.2: Counting Model Scenario.

Figure 5.2 illustrates a partial exploration of the counting model initialized with the same score as above. Notice that for example the scenario ("output", "output") that is possible here, cannot happen in the triggering model.

5.3 Timing

The timing model takes the timing related constraints of the bars into account. Each bar has an offset, duration, a deadline and a slack period that are defined as state variables. All these values determine the earliest start time and the latest start time of a bar and how the corresponding values are set for a triggered bar. This model program also keeps the set of triggers that is set by the Init action; here the mode information is not used, so the triggers map only specifies which bar names are being triggered.

```
namespace Timing {
  partial static class Contract {
    static Map<string, Set<string>> triggers;
    static Set<string> readyBars;
    static double time = 0.0;
    static Map<string, double> offset;
    static Map<string, double> duration ;
    static Map<string, double> deadline;
    static Map<string, double> slack;
  }
}
```

The Execute action for a bar is enabled if the current time is in between the earliest and the latest start times of the bar. The definitions below are assumed to be in the Timing.Contract class.

```
[Action]
static void Execute(string b)
{
  time = time + duration[b];
  readyBars =  readyBars.Remove(b).Union(triggers[b]);
  foreach (string a in readyBars) deadline = deadline.Override(a, time);
}
static bool ExecuteEnabled(string b)
{
  if (!readyBars.Contains(b)) return false;
  double latest = deadline[b] + offset[b] - duration[b];
  double earliest = latest - slack[b];
  return (earliest <= time && time < latest);
}
```

Since triggered bars cannot start before their earliest start time, there is an Idle action which "fast forwards" the time when no triggered action is ready to execute.

```
[Action]
static void Idle()
{
  double ff = (readyBars.IsEmpty ? time : double.MaxValue);
  foreach (string b in readyBars)
    ff = Math.Min(ff, deadline[b] + offset[b] - duration[b] - slack[b]);
  time = ff;
}
static bool IdleEnabled()
{
  foreach (string b in readyBars)
    if (time >= deadline[b] + offset[b] - duration[b] - slack[b]) return false;
  return true;
}
```

A timing model scenario is illustrated in Figure 5.3. The initial values are taken from the sample score used earlier. Notice for example that the first enabled action after Init is Idle, because the time is 0 in this state and the earliest start time for "read" is nonzero.

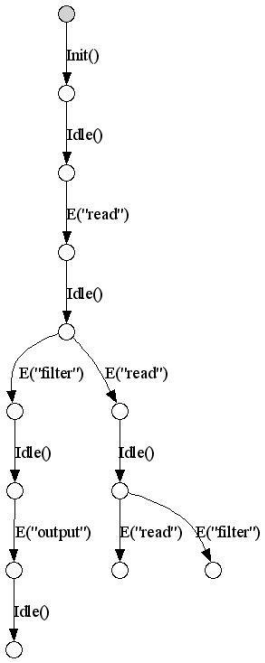

Figure 5.3: Timing Model Scenario.

5.4 Composition of Models

When two model programs A and B are composed into A||B, the action vocabulary of A||B is the union of the action vocabularies of A and B. The state of A||B is the disjoint product of the states of A and B. In other words, the state of A||B has state variables of both A and B. The model programs A and B synchronize through the common actions and interleave all the other actions. When an action *a* is present in the vocabulary of A, but not in the vocabulary of B, then *a* changes only the state variables in A.

The composition of model programs is *syntactic*. It is effectively a program transformation that combines two or more model programs into a new model program. The composed model program has useful algebraic properties. The set of traces of the composed model program is the intersection of the set of traces of the constituent model programs. In particular, one can reason about the action traces and use classical theory of labeled transition systems [10][11].

Figure 5.4 illustrates a partial exploration of the composed model program Triggering||Counting||Timing for the initial values set according to the sample score introduced before. The three model programs synchronize on the Explore and Init actions. The Idle action appears only in the Timing model and is considered as a self-loop in the other models. While the partial model programs can be analyzed separately for conditions related to the corresponding aspects of the scheduler, the composed model program can be analyzed for feature interaction and global safety and liveness conditions.

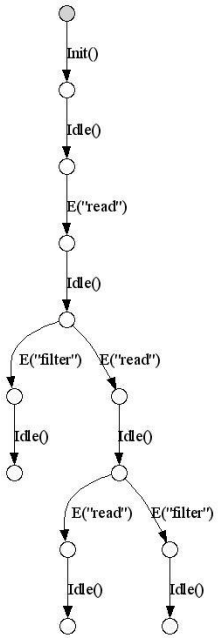

Figure 5.4: Composition Scenario.

6. Adapting distributed partitures

Since the partiture can be viewed as data it is possible to send partitures around the network in messages. This makes it possible for one device to coordinate related activities that involve multiple devices. A partiture that has been adapted on one machine is sent to a worker node that does an admission check and then feeds it to its own scheduler.

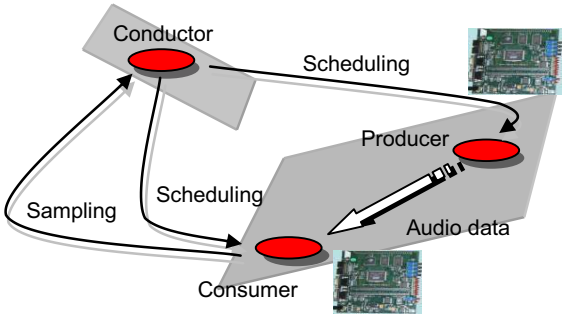

Figure 6.1: Scheme of the stochastic planner in action.

We apply simple stochastic methods to measured execution times. The planning node (the conductor in figure 6.1) predicts the time required to perform a simple program based on past performance. This was

demoed on a test platform equipped with a 25 MHz Arm7 microcontroller with 256KB of ROM and 32KB of RAM. In the core of the demo was a stochastic planner that used the monitored execution times of scheduled functions to make adjustments to the scheduling pattern of the functions.

Initially the planner uses an application supplied fixed schedule for scheduling the jobs on the worker nodes. The schedule is adjusted according to the information on the actual execution times received from worker nodes. Figure 6.2 visualizes the schedule at different points in time, where newer schedules are above older ones. Each line shows the bars for the producer and the consumer, with message transmission times in-between. The schedule is adapted until it reaches a steady state. Once a steady state has been reached the schedule will remain the same until the environment changes.

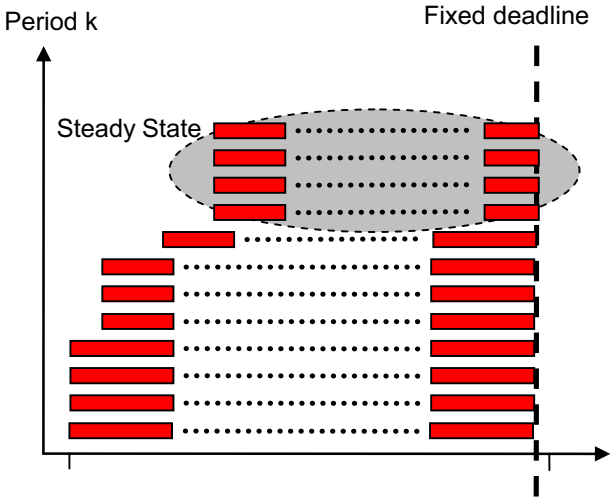

Figure 6.2: Adaptation of bar durations from initial defaults to a near optimal steady state.

6.1 Performance of distributed scheduling

The working of the stochastic planning conductor was estimated through sampling. A simple test method does 20000 multiplications. Starting with no context information the conductor uses an application provided guess.

Once the conductor receives samples from the measured execution times it uses the information with smoothing between each step. The calculation times include formatting and sending the reply message. The table below (figure 6.3) contains the relevant numbers. The estimate is produced by the live conductor, while the mean and deviation have been calculated offline for reference from the raw measurements.

Step	Estimate 95% conf	Measured mean	Standard deviation	Confidence 95%	99%
1	339	337	1.7%	1.0	1.4
2	341	337	1.6%	1.0	1.4
3	346	337	1.8%	1.0	1.4

Figure 6.3: Time measurement and prediction of a CPU intensive task – times in milli-seconds, 32 samples per iteration on embedded microcontroller board. The confidence number indicates the extra time allocated for jitter. Fixed point integer arithmetic rounds the number up slightly.

Since the low-level RTOS scheduler did not produce much jitter, the test was also executed on a PC running Windows XP with the XML communications middleware stack on top. Running without an underlying real-time scheduler introduces more uncertainty but the conductor still deals with it correctly and produces a larger confidence allocation to cope with the increased jitter. As the CPU is faster a million multiplications is done each time. From a steady state the number of calculations is dropped to half. The table below (figure 6.4) shows how the prediction adapts to the drop. The conductor adapts to the larger jitter by padding the estimates.

Step	Estimate 99% conf	Measured mean	Standard deviation	Confidence 95%	99%
1	126	123	6.4%	1.9	2.5
2	124	120	14%	4.2	5.5
3	69	55	2.1%	2.8	3.7
4	58	55	2.9%	3.9	5.2

Figure 6.4: Time measurement on PC in milli-seconds. After the steady state at step 2, the workload is cut in half and the estimate adapts to the new load.

7. Related work

Futures [12] were originally proposed in the Lisp community as a way of deferring evaluation and increasing performance. They were used as a primary construct for concurrency and synchronization in MultiLisp [15]. Later futures have been implemented in mid-level languages, such as Java [21] or C#. To our best knowledge ours is the first native implementation of futures in C on microcontrollers. As an underlying

abstraction for RPC mechanisms it is mentioned in [16], even though not as a primary mechanism.

The partiture idea is first mentioned in [2] by the first author for the purposes of orchestrating distributed embedded web services [1]. The bar is modeled after constraint-based scheduling [19], itself an extension of earliest deadline first [20]. We extend the constraint scheduling by allowing arbitrary schedules and do not limit the resource management to rate-monotonic [17] tasks. We have automated the adaptation of estimates and lifted the chore from application programs. Applying control theory has certainly been used in scheduling, e.g. in Spring [18] but using stochastic processes with constraint scheduling and distributed scheduling is new to our model. The level of parallelism required by an application has been explored in [13] in terms of the *Q model*. Temporal behavior is also studied in [24] in terms of *logical execution time,* the recurring time from the beginning to an end of a process as seen by an independent observer. A separation of a scheduling virtual machine and an execution virtual machine is used in [23], where *schedule-carrying code* allows some scheduling to be done offline, such as verification for non-preemptive scheduling on a given platform. Schedulability for pre-emptive and serialized schedulers is analyzed in *preemption threshold scheduling* [25][26].

As our modeling language we use *model programs* written in C# within the *NModel* framework [4] that is freely available for download [7]. Model programs written in this style are also used in model-based testing and analysis tools like SpecExplorer [9][8], see also [5] for an overview of such modeling techniques and related tools.

8. Conclusion

This paper presented a systematic way to write embedded applications using high-level abstractions, allowing the implementation to be automatically scaled to the resources and level parallelism available on a device, with negligible overhead. On the low end the resulting system is extremely small, a few hundred bytes of ROM and no RAM, while on the high end the program runs on multiple cores in parallel. An application is split into two programs: an implementation and a temporal process description, called a partiture. Futures are used as the programming subtrate for expressing concurrency. The model programs enable safety analysis through bounded exploration and make it possible to derive valid schedules from a given score. The models also facilitate visualization, testing, and conformance checking of various aspects of the expected behavior.

The combination of high-level program expression with tight implementations and sophisticated offline tools created a system that is practical and efficient, while setting up the stage for vastly increased programmer productivity and code reuse. The paper included scenarios for hard real-time programming, distributed adaptive real-time, and expressions of concurrency. The system has been implemented on multiple platforms and measurements were included.

9. References

[1] Helander J., Deeply Embedded XML Communication: Towards an Interoperable and Seamless World, Proceedings of the *5th ACM international conference on Embedded software*, Jersey City, NJ, September 2005.

[2] Helander J., Sigurdsson S., Self-Tuning Planned Actions: Time to Make Real-Time SOAP Real, Proceedings of the *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Seattle, May 2005.

[3] Helander, J., Interoperation and Adaptation for Real-World Computing, *National Workshop on Cyber-Physical Systems*, Alexandria, Virginia, 2006.

[4] Jacky, J., Veanes, M., Campbell, C., and Schulte, W., *Model-based Software Testing and Analysis with C#*, Cambridge University Press, 2007.

[5] Utting, M., and Legeard, B*., Practical Model-Based Testing - A tools approach*, Morgan and Kaufmann, 2006.

[6] Veanes, M., Campbell, C., and Schulte, W., Composition of Model Programs, *FORTE 2007*, *LNCS*, vol. 4574, pp. 128-142, 2007.

[7] NModel, http://www.codeplex.com/NModel, 2007.

[8] SpecExplorer, http://research.microsoft.com/specexplorer, 2006.

[9] Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., and Nachmanson, L., Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, chapter in *Formal Methods and Testing*, Hierons, Bowen and Harman (Eds.), Springer, 2007 (Forthcoming, preliminary version: MSR technical report MSR-TR-2005-59).

[10] Keller, R., Formal verification of parallel programs, *Communications of the ACM*, July, pp. 371—384, 1976.

[11] Lynch, N., and Tuttle, M., Hierarchical correctness proofs for distributed algorithms, *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pp. 137--151, ACM Press, 1987.

[12] Forin, A. (1990) “Futures” in book “Topics in Advanced Language Implementation” Peter Lee ed., MIT Press, Cambridge MA. ISBN: 0-262-12151-4

[13] Mõtus, L. and Rodd, M. “Timing Analysis of Real-time Software”, Elsevier Science/Pergamon, 1994, 212pp

[14] Friedman, D., and Wise, D., Technical Report TR44: CONS should not Evaluate its Arguments (Jan 1976), 26

pages plus appendix [I. S. Michaelson and R. Milner (eds.), Automata, Languages and Programming, Edinburgh University Press, Edinburgh (1976), 256--284]

[15] Halstead, R., Multilisp: A Language for Concurrent Symbolic Computation ACM Transactions on Programming Languages and Systems, Vol 7, No. 4, October 1985

[16] Walker, E., Floyd, R., Neves, P., Asynchronous remote operation execution in distributed systems, 10th International Conference on Distributed Computing Systems, Paris, France, 1990

[17] Mercer, C., Rajkumar, R., and Tokuda, H., Applying hard real-time technology to multimedia systems. In Proceedings of the Workshop on the Role of Real-Time in Multimedia/Interactive Computing Systems, November 1993.

[18] Stankovic, J., Lu, C., Son, S., Tao, G., The case for feedback control real-time scheduling, Proceedings of the 11th Euromicro Conference on Real-Time Systems,York, UK, 1999.

[19] Jones, M., Roşu, D., Roşu, M., CPU reservations and time constraints: efficient, predictable scheduling of independent activities, Proceedings of the sixteenth ACM symposium on Operating systems principles, Saint Malo, France, 1997.

[20] Liu, C., Layland, J., Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Journal of the ACM, New York, USA, 1973.

[21] Cugola, G. and C. Ghezzi, CJava: Introducing concurrent objects in Java, in 4th International Conference on Object Oriented Information Systems, 1997.

[22] Agrawal, G., Acharya, A., and Saltz, J. An interprocedural framework for placement of asynchronous I/O operations. In Proceedings of the 10th international Conference on Supercomputing, Philadelphia, USA, May 1996.

[23] Henzinger, T., Kirsch, C., and Matic, S., Schedule-carrying code. In Proceedings of EMSOFT, Philadelphia, USA, October 2003. LNCS 2855, pp. 241--256, Springer, 2003.

[24] Farcas, E., Farcas, C., Pree, W., and Templ J., Transparent distribution of real-time components based on logical execution time, SIGPLAN Notices, Volume 40 Issue 7, pp. 31-39, 2005.

[25] Wang, Y., Saksena, M., Scheduling fixed-priority tasks with preemption threshold, Real-Time Computing Systems and Applications (RTCSA), Hong Kong, December, 1999.

[26] Regehr, J., Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults, RTSS 2002, Austin, USA, December, 2002.