Event-Pattern Triggered Real-Time Tasks[*]

Jan Carlson, Jukka Mäki-Turja and Mikael Nolin
Mälardalen Real-Time Research Centre (MRTC), Västerås, Sweden
jan.carlson@mdh.se

Abstract

We present the concept of pattern-triggered tasks which are released when a particular pattern of events occur. A formal event algebra is used to define complex triggering conditions for these tasks, and the detection of triggering conditions is performed within the system by code generated automatically from these definitions. The implementation of the algebra has many desirable features for resource constrained real-time systems, including bounded and low execution time and memory consumption. Furthermore, we present novel schedulability analysis for our pattern-triggered tasks that leverage on existing analysis for fixed-priority and dynamic-priority scheduling policies.

1. Introduction

In this paper we present the concept of *pattern-triggered tasks*. Pattern-triggered tasks are released for execution when a, potentially complex, pattern of events occur. The events and the triggering conditions are formally specified by an event algebra that allow complex combinations of events to be expressed [7]. To guarantee bounded response-times and make our systems amenable for scheduling analysis, the individual events that build up these event patterns are assumed to be sporadic with a known minimum interarrival time.

Our event algebra has attractive properties making it suitable for engineering of resource constrained real-time systems. First and foremost, the algebra provides simple and intuitive operators (e.g., conjunction, disjunction and sequence). Furthermore, the code for pattern detection can be automatically generated from the event expressions, and exhibits necessary qualities in order to be used in resource constrained real-time systems: (i) it is resource efficient, and bounded, with regards to memory footprint as well as processing time; (ii) worst case execution time (WCET) can easily be analysed by automated tools since the code only contains simple programming constructs.

Pattern-triggered tasks can be realised without explicit support for pattern detection in the underlying real-time operating system. Moreover, they can be modelled as a particular set of sporadic tasks, which allows traditional schedulability analysis techniques for sporadic task models under fixed priority (FPS) and earliest deadline first (EDF) scheduling policies, to be applied. Thus, the proposed technique allows co-existence of periodic, sporadic and pattern-triggered tasks in a single system and that high precision schedulability results can be obtained for all three types of tasks.

Example As a running example throughout the paper, we consider a system where the external events are a button, a pressure alarm and a temperature alarm, and where the desired behaviour includes performing a particular response when both alarms have occurred, unless the button is pressed in between.

The rest of the paper is organised as follows: Section 2 surveys related work, and Section 3 defines the task model and describes how pattern-triggered tasks are realised. Section 4 presents the event algebra used to define event patterns, including formal semantics and important properties. The generation of detection code is described in Section 5, and in Section 6 we present schedulability analysis for task sets including pattern-triggered tasks, before concluding the paper in Section 7.

2. Related Work

The concept of pattern-triggered tasks is quite different from the concept of event-streams that are used to model chains of tasks by, e.g., Chakraborty *et al.* [8] and in SymTA/S [13]. In short, they consider a scenario where tasks are triggered by external events that can have complex occurrence patterns, while in our approach the external events are simple and the detection of significant event patterns is performed within the system. Common to the two approaches is that information about event patterns is exploited in the schedulability analysis to achieve more accurate results.

Timed automata has been proposed as a modelling language for complex triggering patterns [3]. However, for practically oriented engineers it is often a daunting task to specify system behaviour using timed automata. Furthermore, the computational complexity of analysing such

[*]This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS and by the Knowledge Foundation within the project MultEx.

specifications with respect to schedulability can be overwhelming. In particular, constructs that require that multiple patterns are detected concurrently, such as conjunction or non-occurrence of a complex pattern, typically result in very large automata compared to the size of the automata detecting the constituent parts [22].

More directly related to our approach, Mok *et al.* [20] present a framework based on real time logic (RTL), which is a first order logic with a dedicated predicate encoding event occurrences. The framework has, for example, been used in the context of network management [18] and in an electronic brokerage application [21]. It has also been suggested as the basis for composite event specification in active databases [17]. Bounded detection algorithms are presented, but the memory bound depends on the time constants used in the pattern definitions and on the minimum interarrival time of events [20].

The operators of our algebra are influenced by work in the area of active databases [9, 11, 12]. The use of interval semantics to avoid unintended semantics for some operation combinations was proposed by Galton and Augusto [10]. Our restriction policy, used to achieve resource bounds, resembles the concept of event contexts found in e.g., Snoop [9] and Solicitor [19], but is different in that we apply it once to the whole expression rather than to each operator individually.

3. Event Pattern Triggered Tasks

A straightforward way to construct an embedded system that reacts to a particular event pattern is to include a task which performs pattern detection implicitly within the code and then performs the desired reaction if the pattern is detected.

The drawback of this straightforward approach is that the execution time of the task varies a lot, since the response code is only executed when the full pattern is detected. If pattern occurrences are rare compared to the occurrences of individual events, analysis techniques based on a single WCET value will be very pessimistic, meaning that the system must be significantly underutilised in order to statically guarantee timeliness of all tasks.

Alternatively, pattern detection and response can be split into two tasks. This allows tighter WCET information, but does not solve the utilisation problem. With no information about the nature of the pattern, it must be assumed that the response task is executed as often as the detection task.

We propose a task model where triggering patterns are defined explicitly rather than implicitly in the task code. This separation of concerns facilitates both design and analysis. Complex event patterns can be specified on a high level of abstraction, independently from the rest of the application in which they are used, and the rest of the system is free from detection related code and information about partially completed patterns.

With a formal syntax and semantics, the detection code can be automatically generated from pattern specifications, relieving the programmer of this error prone task. Also, the patterns are available for analysis, in particular to establish how frequently the pattern can occur in the worst case, which allows a more efficient use of resources compared to assuming that the complicated event occurs as often as its constituent simple events.

3.1. Example task model and assumptions

For the sake of presentation, we adopt a relatively simple task model including both periodic tasks and tasks that are triggered by particular patterns of sporadic events. The system is fully preemptive and scheduled according to either EDF or FPS. Ordinary sporadic tasks are treated as a special case of pattern-triggered tasks where the pattern consists of just a single event. Extending the task model to include precedence constraints, blocking caused by shared resources, etc., should be straightforward.

A task, τ_i, is characterised by the following parameters:

- **Worst case execution time** (C_i) The longest time it could take to execute the code of the task, assuming that it is not interrupted.

- **Relative deadline** (D_i) The time, relative to activation, when the task must be finished.

Each task also has one of the following parameters, depending on if it is periodic or pattern-triggered:

- **Period** (T_i) The time between two consecutive activations of a periodic task.

- **Event expression** (E_i) A specification of the situation under which a pattern-triggered task should be activated.

Note that for pattern-triggered tasks, C_i refers to the execution time of the user provided response code. The worst case execution time associated with the generated detection code of an event expression E_i is denoted by $\mathrm{wcet}(E_i)$.

Under FPS, the following parameter also applies:

- **Priority** (P_i) The static priority of the task.

Task instances that have the same priority are assumed to be served in the same order as they are released. Ties are broken arbitrarily.

Example For the running example, we let the system consist of three tasks, two of which are periodic (τ_1 and τ_3). The third task, τ_2, is triggered by the event pattern *“both alarms (*P *and* T*) occur without the button* B *being pressed in between”*. In the event algebra, elaborated in Section 4, this situation is specified by the event expression $(\mathrm{P}+\mathrm{T})-\mathrm{B}$. The parameters of this example task set are:

Task	P_i	C_i	T_i	D_i	E_i
τ_1	High	10	50	30	–
τ_2	Mid	20	–	100	$(\mathrm{P}+\mathrm{T})-\mathrm{B}$
τ_3	Low	30	200	200	–

3.2. Realisation

Allowing pattern-triggered tasks to be directly used in an application would require an underlying real-time operating system (RTOS) with pattern detection capabilities. Instead, we propose an indirect approach where pattern-triggered tasks are realised by constructs found in traditional RTOSes.

In the concrete system, the response code of a pattern-triggered task is merged with code for detecting the event pattern, automatically generated based on the detection algorithm presented in Section 5. Together, they form an event triggered task, which is triggered individually by all events in the pattern specification.

When an event occurs, the interrupt handler activates all tasks with patterns including that event, providing the ID of the event and a time stamp in the activation. When selected for execution by the scheduler, the task first executes the detection code with the event ID and time stamp received from the interrupt handler at activation. If the detection algorithm signals a successful detection of the whole pattern, the task proceeds with executing the response code, otherwise it terminates. Figure 1 shows the realisation of the pattern triggered task τ_2 from the running example.

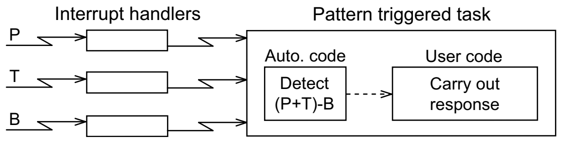

Figure 1. Realisation of the pattern-triggered task τ_2 triggered by the pattern $(\mathrm{P}+\mathrm{T})-\mathrm{B}$.

In a situation when several events occur before the first has been processed, multiple instances of a pattern-triggered task will be active at the same time. Under FPS, they have the same priority, but the “first-come-first-served” assumption guarantees that the events are processed in the correct order. For EDF, this is ensured by the fact that the absolute deadlines of the instances follow the order in which the events arrived.

The reason for including the conceptually different activities of event detection and response in the same task is related to the fact that the original deadline of a pattern-triggered task is relative to the pattern occurrence, not the time at which the pattern is detected [4]. If detection and response were handled by separate tasks, the response task could not be statically assigned a deadline relative to its activation, which is prescribed by the traditional task model.

4. The Event Algebra

This section presents the event algebra used to specify triggering patterns, focusing on aspects that are most relevant to the scope of this paper. For an in-depth treatment, including examples motivating the use of interval semantics, comparisons with alternative approaches, and formal proofs of properties, complexity and correctness, see [6].

The particular characteristics of this event algebra is the combination of satisfying a number of intuitive algebraic laws while at the same time permitting any event expression to be correctly detected with bounded memory.

4.1. Syntax

We denote by $\mathcal{P}$ the set of external events that the system can react to. In the algebra, they are *primitive event expressions*, representing the simple pattern of a single event occurring. For more complex patterns, *composite event expressions* can be built using the operators of the algebra.

Definition 1 *If $A \in \mathcal{P}$, then A is an event expression. If A and B are event expressions, and if t is a time constant, then $A \vee B$, $A + B$, $A ; B$, $A - B$ and A_t are event expressions.*

Informally, a *disjunction* $A \vee B$ represents that either of A and B occurs. A *conjunction* means that both events have occurred, in any order and possibly at different times, and is denoted $A+B$. A *sequence* $A;B$ is an occurrence of A followed by an occurrence of B. The *negation*, denoted $A-B$, occurs when there is an occurrence of A during which there is no occurrence of B. This description of negation makes little sense when A is primitive, since primitive occurrences are instantaneous. A more typical usage of the negation operator is $(A;B)-C$, representing that there should be no occurrence of C between the occurrences of A and B. Finally, there is a *temporal restriction* A_t which occurs when there is an occurrence of A shorter than t time units. As with negation, the temporal restriction is typically applied to sequences or conjunctions, restricting the time between the constituent occurrences.

Example For the example system, we have $\mathcal{P} = \{\mathrm{B}, \mathrm{P}, \mathrm{T}\}$ and the triggering pattern is specified by the event expression $(\mathrm{P}+\mathrm{T})-\mathrm{B}$. For a more complex example, consider the pattern *“the button is pressed twice within two seconds, and none of the alarms occurs in between”* which can be defined by the event expression $(\mathrm{B};\mathrm{B})_{2\,\mathrm{sec}}-(\mathrm{P}\vee\mathrm{T})$.

4.2. Semantics

We represent a single event occurrence by its start- and end time. Although primitive events are assumed to be instantaneous, occurrences of composite events must be represented by intervals in order to achieve some of the desired algebraic properties [6].

Formally, the interval of an event occurrence a is captured by the functions $\mathrm{start}(a)$ and $\mathrm{end}(a)$, where $\mathrm{end}(a)$ denote the time of occurrence, and the full interval from $\mathrm{start}(a)$ to $\mathrm{end}(a)$ represents the smallest interval containing all primitive occurrences that caused a.

The operator $\oplus$ is used to construct occurrences of composite event expressions from occurrences of the constituents. E.g., each occurrence of $A;B$ will be constructed from one occurrence of A and one occurrence of B. The start- and end time of composite occurrences are defined as follows:

$$\begin{aligned} \mathrm{start}(a \oplus b) &= \min(\mathrm{start}(a), \mathrm{start}(b)) \\ \mathrm{end}(a \oplus b) &= \max(\mathrm{end}(a), \mathrm{end}(b)) \end{aligned}$$

The following definition formalises the algebra semantics:

Definition 2 *For $A \in \mathcal{P}$, $[\![A]\!]$ is the set of all occurrences of A. For composite event expressions, the semantics is defined as follows:*

$$\begin{aligned} [\![A \vee B]\!] &= [\![A]\!] \cup [\![B]\!] \\ [\![A + B]\!] &= \{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!]\} \\ [\![A;B]\!] &= \{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b)\} \\ [\![A - B]\!] &= \{a \mid a \in [\![A]\!] \wedge \neg \exists_{b \in [\![B]\!]}(\mathrm{start}(a) \leq \mathrm{start}(b) \wedge \\ &\qquad \mathrm{end}(b) \leq \mathrm{end}(a))\} \\ [\![A_t]\!] &= \{a \mid a \in [\![A]\!] \wedge \mathrm{end}(a) - \mathrm{start}(a) \leq t\} \end{aligned}$$

This definition results in an algebra with simple semantics and intuitive algebraic properties. However, the memory required to perform the detection is unbounded for some expressions. For example, when detecting $A;B$ each occurrence of A must be remembered forever, since it should be combined with all future occurrences of B. To deal with resource limitations, we introduce a formal restriction policy that defines a subset of occurrences that must be detected. The basic idea is to ignore simultaneous occurrences, while at the same time retaining the desired properties of the semantics.

The restriction policy is defined as a binary relation *res*. Rather than computing the full $[\![A]\!]$ for a given event expression A, an implementation of the algebra is expected to detect a set of event occurrences S such that *res*(A, S) holds.

Informally, from the occurrences with the same end time, exactly one with maximal start time must be detected. For the user of the algebra, this means that at any time when there is one or more occurrences of A, one of them will be detected.

Definition 3 *For an event expression A and a set S of event occurrences, res(A, S) holds if:*

1. $S \subseteq [\![A]\!]$
2. $\forall_{a \in [\![A]\!]}(\exists_{s \in S}(\mathrm{start}(a) \leq \mathrm{start}(s) \wedge \mathrm{end}(a) = \mathrm{end}(s)))$
3. $\forall_{s,s' \in S}\ (\mathrm{end}(s) = \mathrm{end}(s') \Rightarrow s = s')$

4.3. Properties

Next, we present a selection of algebraic laws showing to what extent the operators behave according to intuition. Moreover, the laws facilitate formal and informal reasoning about the algebra and about tasks triggered by patterns defined by the algebra. For example, they could form a basis for rewriting event expressions to minimise the resources required for detection.

For this, we first define expression equivalence as follows:

Definition 4 *For two event expressions A and B we define $A \equiv B$ to hold if $[\![A]\!] = [\![B]\!]$ independently of how the primitive events occur.*

Now, the laws can be formulated. For proofs and a more extensive set of laws, see [6].

Laws 1 *For arbitrary event expressions A, B and C, and any time constant t, the following laws hold:*

$$\begin{aligned} A \vee A &\equiv A & (A_t)_{t'} &\equiv A_{\min(t,t')} \\ A \vee B &\equiv B \vee A & A \vee (B \vee C) &\equiv (A \vee B) \vee C \\ A + B &\equiv B + A & A + (B + C) &\equiv (A + B) + C \\ A_t &\equiv A \quad \textit{if } A \in \mathcal{P} & A;(B;C) &\equiv (A;B);C \end{aligned}$$

Laws 2 *For arbitrary event expressions A, B and C, and any time constant t, the following laws hold:*

$$\begin{aligned} (A \vee B) + C &\equiv (A + C) \vee (B + C) \\ (A \vee B);C &\equiv (A;C) \vee (B;C) \\ A;(B \vee C) &\equiv (A;B) \vee (A;C) \\ (A - B) - C &\equiv A - (B \vee C) \\ (A \vee B) - C &\equiv (A - C) \vee (B - C) \\ (A \vee B)_t &\equiv A_t \vee B_t \\ (A - B)_t &\equiv A_t - B \end{aligned}$$

These laws identify expressions that are semantically equivalent, but in order to deal with resource limitations, we expect an implementation of the algebra to compute an event stream S such that *res*(A, S), rather than computing $[\![A]\!]$. As a result, detecting A might potentially yield a different result than detecting B, even when $A \equiv B$. Consequently, it should be clarified to what extent the laws presented above are applicable when the restriction policy is applied.

Theorem 1 *If $A \equiv B$ and res(A, S), then res(B, S).*

Proof: Since $A \equiv B$ implies that $[\![A]\!] = [\![B]\!]$, this holds trivially. □

Thus, $A \equiv B$ ensures that for any implementation consistent with the restriction policy, the occurrences detected for A are always a valid result for B as well. Thus, any reasoning based on the algebra semantics and restriction policy, and not on details of a particular detection algorithm, will be equally valid for equivalent expressions.

5. Implementation

A primary ambition when designing the algebra has been to allow an implementation for which bounds on memory footprint and processing time can be determined statically. This section presents how a concrete online detection algorithm can be generated automatically from an event expression. The detection algorithm has a memory and time complexity of O(m^2) where m is the size of the expression.

This is an improved version of the algorithm compared to what has been previously published [7], where bounds on memory and time could only be established for a subset of expressions.

5.1. Detection algorithm

Figure 2 presents an algorithm that detects occurrences of a given event expression A. The numbers $1 \ldots m$ are assigned to the subexpressions of A in an arbitrary bottom-up order, and we let A^i denote subexpression number i. Consequently, we have $A^m = A$.

The symbol ε is used to represent a non-occurrence, and we define $\text{start}(\varepsilon) = \text{end}(\varepsilon) = -1$ to simplify the algorithm. The algorithm is executed every time tick when at least one primitive event occurs, and at the end of execution, the variable a_m contains the current occurrence of A, or ε if A did not occur.

The variables used in the algorithm can be divided into three categories. *Persistent* variables (l, r, Q and t) store information that must be remembered from one time tick to the next in order to detect the event properly. Since each subexpression requires its own persistent variables, they are indexed from 1 to m.

Auxiliary variables (a and S) are used to pass information from a subexpression to its parent node in the expression tree, and are indexed in the same way as the persistent variables. In short, a_i is used to store the current instance of subexpression A^i, and S_i contains possible start times of future occurrences of A^i.

There are also *temporary* variables (t, e, e' and Q') that are used locally within a single subexpression in a single tick. These are not indexed, indicating that the content is never used outside that scope.

Persistent and auxiliary variables are initialised as follows: $t_i = -1$, $l_i = r_i = \varepsilon$ and $S_i = Q_i = \emptyset$ for $1 \leq i \leq m$.

```
for $i$ from $1$ to $m$
  if $A^i \in \mathcal{P}$ then
    $a_i$ := the current occurrence of $A^i$.
  if $A^i = A^j \vee A^k$ then
    if $\text{start}(a_j) \leq \text{start}(a_k)$ then $a_i := a_k$
    else $a_i := a_j$
    $S_i := S_j \cup S_k$
  if $A^i = A^j + A^k$ then
    if $\text{start}(l_i) < \text{start}(a_j)$ then $l_i := a_j$
    if $\text{start}(r_i) < \text{start}(a_k)$ then $r_i := a_k$
    if $l_i = \varepsilon \vee r_i = \varepsilon \vee (a_j = \varepsilon \wedge a_k = \varepsilon)$ then $a_i := \varepsilon$
    else if $\text{start}(a_k) \leq \text{start}(a_j)$ then $a_i := a_j \oplus r_i$
      else $a_i := l_i \oplus a_k$
    $S_i := S_j \cup S_k \cup \{\text{start}(l_i),\ \text{start}(r_i)\} \backslash \{-1\}$
  if $A^i = A^j ; A^k$ then
    $e' := \varepsilon$
    foreach $e$ in $Q_i \cup \{l_i\}$
      if $\text{end}(e) < \text{start}(a_k) \wedge \text{start}(e') < \text{start}(e)$
      then $e' := e$
    if $e' \neq \varepsilon$ then $a_i := a_k \oplus e'$ else $a_i := \varepsilon$
    $Q' := \emptyset$
    foreach $t$ in $S_k$
      $e' := \varepsilon$
      foreach $e$ in $Q_i \cup \{l_i\}$
        if $\text{end}(e) < t \wedge \text{start}(e') < \text{start}(e)$
        then $e' := e$
      $Q' := Q' \cup \{e'\}$
    $Q_i := Q'$
    if $\text{start}(l_i) < \text{start}(a_j)$ then $l_i := a_j$
    $S_i := S_j \cup \{\text{start}(e) \mid e \in Q_i \cup \{l_i\}\} \backslash \{-1\}$
  if $A^i = A^j - A^k$ then
    if $t_i < \text{start}(a_k)$ then $t_i := \text{start}(a_k)$
    if $t_i < \text{start}(a_j)$ then $a_i := a_j$ else $a_i := \varepsilon$
    $S_i := S_j$
  if $A^i = (A^j)_t$ then
    if $\text{end}(a_j) - \text{start}(a_j) \leq t$ then $a_i := a_j$
    else $a_i := \varepsilon$
    $S_i := S_j$
```

Figure 2. General detection algorithm.

Theorem 2 *Assume that the algorithm is executed every tick when some primitive event occurs, and let S denote the set of detected occurrences, i.e., the contents of a_m after executing the algorithm, except when $a_m = \varepsilon$. Then res(A, S) holds.*

Proof: A detailed formal proof is given in [6]. □

```
$a_1$ := the current occurrence of P
$a_2$ := the current occurrence of T
**if** $\text{start}(l_3) < \text{start}(a_1)$ **then** $l_3 := a_1$
**if** $\text{start}(r_3) < \text{start}(a_2)$ **then** $r_3 := a_2$
**if** $l_3 = \varepsilon \vee r_3 = \varepsilon \vee (a_1 = \varepsilon \wedge a_2 = \varepsilon)$ **then** $a_3 := \varepsilon$
**else if** $\text{start}(a_2) \leq \text{start}(a_1)$ **then** $a_3 := a_1 \oplus r_3$
  **else** $a_3 := l_3 \oplus a_2$
$a_4$ := the current occurrence of B.
**if** $t_5 < \text{start}(a_4)$ **then** $t_5 := \text{start}(a_4)$
**if** $t_5 < \text{start}(a_3)$ **then** $a_5 := a_3$ **else** $a_5 := \varepsilon$
```

Figure 3. Concrete detection algorithm for (P+T)−B.

5.2. Code generation

The detection algorithm in Figure 2 is formulated for arbitrary expressions, and the main loop selects dynamically which part of the algorithm to execute for each subexpression. When the expression is known at compile-time, the main loop can be unrolled and the top-level conditionals, as well as all indices, can be statically determined. Also, assignments of S_i variables can be removed for all subexpressions except those appearing somewhere in the second argument of a sequence operator.

Example The concrete detection algorithm for the event expression in the example system is presented in Figure 3. Initially, $t_5 = -1$ and $l_3 = r_3 = \varepsilon$.

5.3. Time and memory requirements

The worst part of the algorithm, from a complexity point of view, is the nested foreach constructs in the sequence part. However, this source of complexity can be avoided, without compromising the correctness of the algorithm, if the set variables S_i and Q_i are represented as ordered structures. This means that the nested foreach constructs can be changed into a single while-loop traversing S_i and Q_i together. For details, and for the proof of the following theorem, see [6].

Theorem 3 *The time and memory complexity of the algorithm is* $O(m^2)$*, where* m *denotes the number of subexpressions in* A*.*

The generated code does not utilise any dynamic memory management, neither by explicit memory allocation, nor by function calls or parameter passing via the runtime stack. Furthermore, the code is characterised by a very simple control flow. For example, there are no subroutine or function calls, and all loops are trivially bounded by the size of some static data structure. Thus, existing techniques and standard tools for execution time analysis, e.g., SWEET [28], aiT [1] or Bound-T [5] should be applicable.

6. Schedulability Analysis

For the analysis we assume that events are sporadic, i.e., that the minimum interarrival time, denoted $\text{mint}(A)$ is known for each $A \in \mathcal{P}$.

Example Before analysing the example task set, we make the following assumptions about minimum interarrival times and worst case detection execution time: $\text{mint}(\text{P}) = 70$, $\text{mint}(\text{T}) = 200$, $\text{mint}(\text{B}) = 60$ and $\text{wcet}((\text{P}+\text{T})-\text{B}) = 5$.

With the straightforward approach described initially, where pattern detection is performed implicitly within the task code, this task set would be considered non-schedulable even if the detection overhead is disregarded. During any time interval of length 4200, 84 instances of τ_1 and 21 instances of τ_3 are released for execution. Furthermore, there can be 60, 21 and 70 occurrences of P, T and B, respectively, potentially triggering 151 instances of τ_2. Thus, the total amount of execution that is released during the interval is $84 * 10 + 21 * 30 + 151 * 20 = 4490$. Since 4490 units of computation can be released in any interval of length 4200, the task set is clearly non-schedulable.

Introducing pattern-triggered tasks with explicit pattern specifications, however, permits a more accurate analysis. We will show later that this particular task set is in fact schedulable.

The occurrences of a pattern are in general not sporadic even though the primitive events are. For a simple example, consider two sporadic events P and T. Regardless of the minimum interarrival times of the two events, an occurrence of P can be immediately followed by an occurrence of T, resulting in two occurrences of the pattern P∨T separated by just a single clock tick.

Nevertheless, the frequency at which a pattern occurs can be bounded, but in a more general way than with a single minimum interarrival time. For example, we can safely state that in any interval of length $\min(\text{mint}(\text{P}), \text{mint}(\text{T}))$, there can be at most two occurrences of the pattern P∨T.

This resembles the concept of *bursty aperiodic tasks* [14] which are triggered by events that can occur arbitrarily close in time, but which are bounded by a constraint specifying that there can be at most n occurrences within any interval of length l.

For patterns, however, it is possible to be more specific than the two parameters of bursty tasks permit. First, we note that a pattern always occur at the same time as one of the constituent primitive events. In fact, only a subset of the constituent events can directly result in an occurrence of the pattern. For example, $A;B$ always occur at the same time as some B occurrence. The following definition formalises this idea.

Definition 5 *For an event expression* A*,* $\text{prim}(A)$ *denotes the set of primitive event expressions in* A *and* $\text{term}(A)$ *denotes the set of terminating primitive events of* A*, de-*

fined as follows:

$$\begin{aligned}
\mathrm{term}(A) &= \{A\} \quad \text{if } A \in \mathcal{P} \\
\mathrm{term}(A{\vee}B) &= \mathrm{term}(A) \cup \mathrm{term}(B) \\
\mathrm{term}(A{+}B) &= \mathrm{term}(A) \cup \mathrm{term}(B) \\
\mathrm{term}(A;B) &= \mathrm{term}(B) \\
\mathrm{term}(A{-}B) &= \mathrm{term}(A) \\
\mathrm{term}(A_t) &= \mathrm{term}(A)
\end{aligned}$$

The central idea in the schedulability analysis of systems with pattern-triggered tasks is that a task responding to occurrences of a pattern requires the same amount of computation resources over time as a particular set of sporadic tasks. A pattern-triggered task is realised by one task which is activated by all events in the pattern, as described in Section 3.2. During the schedulability analysis, however, we view task instances that were activated by one event as instances of a different task than those activated by another event. For example, when τ_2 is activated by P, this is viewed as an instance of a separate task than when activated by T or B. This also allows taking into consideration the information provided by the explicit pattern specifications, by treating separately those instances that only ever execute the detection code and never the response code.

Formally, from the original task set of periodic and pattern-triggered tasks, we construct a fictive, *auxiliary task set* which is used for schedulability analysis:

Definition 6 *Given a task set* Γ*, the auxiliary task set* Γ^{aux} *is the smallest set of tasks such that*

- *all periodic tasks in* Γ *are in* Γ^{aux}*;*
- *for each pattern-triggered task* $\tau_i \in \Gamma$*, and each primitive event* $A \in \mathrm{prim}(E_i)$*,* Γ^{aux} *contains a sporadic task* $\tau_{i\mathrm{A}}$ *with* $T_{i\mathrm{A}} = \mathrm{mint}(A)$*,* $D_{i\mathrm{A}} = D_i$*,* $P_{i\mathrm{A}} = P_i$ *and*
$$C_{i\mathrm{A}} = \begin{cases} \mathrm{wcet}(E_i) + C_i & \text{if } A \in \mathrm{term}(E_i) \\ \mathrm{wcet}(E_i) & \text{if } A \notin \mathrm{term}(E_i) \end{cases}$$

Example The auxiliary task set for the running example, constructed according to Definition 6, is shown below:

τ_i	P_i	C_i	T_i	D_i
τ_1	High	10	50	30
$\tau_{2\mathrm{P}}$	Mid	25	70	100
$\tau_{2\mathrm{T}}$	Mid	25	200	100
$\tau_{2\mathrm{B}}$	Mid	5	60	100
τ_3	Low	30	200	200

Since $\mathrm{prim}((\mathrm{P}{+}\mathrm{T}){-}\mathrm{B}) = \{\mathrm{P}, \mathrm{T}, \mathrm{B}\}$, the pattern triggered task τ_2 results in three sporadic tasks in the auxiliary task set ($\tau_{2\mathrm{P}}$, $\tau_{2\mathrm{T}}$ and $\tau_{2\mathrm{B}}$), with minimum interarrival times given by these primitive events, respectively. From $\mathrm{term}((\mathrm{P}{+}\mathrm{T}){-}\mathrm{B}) = \{\mathrm{P}, \mathrm{T}\}$ it follows that the WCET of the response code should be included only in $\tau_{2\mathrm{P}}$ and $\tau_{2\mathrm{T}}$.

Theorem 4 *If* Γ^{aux} *is schedulable then* Γ *is schedulable.*

Proof: We show that any sequence of Γ task instances can be mirrored by a sequence of Γ^{aux} task instances with the same arrival times, priorities, execution times and deadlines. Since the periodic tasks are the same in both task sets, we only need to consider the pattern-triggered tasks.

Every instance of a pattern-triggered task τ_i from Γ is triggered by an occurrence of some event in $\mathrm{prim}(E_i)$. Thus, the activation times of τ_i can be mirrored by activations of the corresponding sporadic tasks in Γ^{aux}.

Next, consider an individual instance of τ_i, triggered by an occurrence of the primitive event A. If $A \notin \mathrm{term}(E_i)$, then an occurrence of A can never result in a full occurrence of the pattern, and thus the τ_i instance only executes the detection algorithm and not the response. This is consistent with the WCET of $\tau_{i\mathrm{A}}$, which is defined as $C_{i\mathrm{A}} = \mathrm{wcet}(E_i)$ when $A \notin \mathrm{term}(E_i)$. For events in $\mathrm{term}(E_i)$, Γ^{aux} safely approximates Γ by assuming that they can result in a full occurrence of the pattern. Clearly, the execution time of the τ_i instance can not exceed $\mathrm{wcet}(E_i) + C_i$.

Altogether, this means that any sequence of Γ task instances can be mirrored by an instance sequence of tasks from Γ^{aux} with the same arrival times, priorities and execution times. Thus, if Γ^{aux} is schedulable, so is Γ. □

The tasks in the auxiliary task set are either periodic or sporadic, which means that existing schedulability analysis theory [24] for EDF and FPS, respectively, can be applied.

Example (EDF) Under the EDF scheduling policy, a given task set is schedulable if and only if no deadline is violated during the busy period. This was shown by Liu and Layland [16] for task sets where $D_i = T_i$, and extended to less restricted task sets by Spuri [26] and Ripoll *et al.* [23] independently. For the example system, we have a busy period of length 190, and the following absolute deadlines must be investigated:

$$\{d \mid d = kT_i + D_i,\ d \le 190,\ \tau_i \in \Gamma^{\mathrm{aux}},\ k \ge 0\} = \{30,\ 80,\ 100,\ 130,\ 160,\ 170,\ 180\}$$

For each deadline in this set, the processor demand $h(d)$ up to that point is computed:

$$\begin{array}{lll}
h(30) = 10 & h(130) = 85 & h(180) = 125 \\
h(80) = 20 & h(160) = 90 & \\
h(100) = 75 & h(170) = 115 &
\end{array}$$

Since $h(d) \le d$ for all deadlines within the busy period, the task set is schedulable under EDF.

Example (FPS) In an FPS context, standard response time analysis [25] can be applied, taking into consideration that deadlines may be larger than periods [15, 27], and that tasks do not have unique priorities. For the example task set, a worst-case response time r_i is computed for each τ_i in the auxiliary task set:

τ_i	P_i	C_i	T_i	D_i	r_i
τ_1	High	10	50	30	10
$\tau_{2\mathrm{P}}$	Mid	25	70	100	75
$\tau_{2\mathrm{T}}$	Mid	25	200	100	75
$\tau_{2\mathrm{B}}$	Mid	5	60	100	75
τ_3	Low	30	200	200	190

Since $r_i \leq D_i$ for all tasks in Γ^{aux}, the original task set Γ is schedulable under FPS.

At first, it may seem that approximating a pattern-triggered task by a set of sporadic tasks introduces a lot of pessimism. However, as a result of the algebra detecting also partially overlapping occurrences of a pattern, this over-approximation only happens for a relatively limited number of expressions. In fact, any expression that does not contain negation or temporal restriction can in the worst case occur exactly as often as the auxiliary taskset counterpart. With negation or temporal restriction, it is possible to construct expressions for which the auxiliary taskset is an over-approximation. For example, P−P or $(\mathrm{P;P})_t$ when $t < \mathrm{mint(P)}$ define patterns which never occur, but which are assumed to occur as often as P in the auxiliary taskset.

One way to reduce the over-approximation in such cases would be to simplify the expression, based on the algebraic laws exemplified in Section 4.3, before constructing the auxiliary taskset.

7. Conclusion and Future Work

This paper introduces the concept of event pattern triggered tasks, where tasks can be triggered by more complex event patterns than the traditional periodic or sporadic event streams, and shows how such tasks can be realised for existing real-time operating systems without support for pattern detection. Furthermore, we propose an event algebra that is able to express the activation criteria of such tasks. The algebra exhibit several properties that make it useful in a resource constrained real-time setting:

- **Algebraic laws** The laws show that the algebra corresponds to what one might intuitively expect from the operators, and facilitates analysis and optimisation of event expressions.

- **Realisation and implementation** The pattern detection code can be automatically generated from the triggering expressions. Moreover, the run-time footprint, in terms of CPU utilisation and memory consumption, is low and statically bounded. The code also exhibits properties that makes automatic WCET calculation by existing tools feasible.

- **Schedulability analysis** Assuming that constituent events are sporadic, the task set can be transformed into a standard periodic/sporadic task model, which means that standard scheduling theory can be applied. If the result of the transformation is schedulable, then so is the original task set.

To simplify the presentation, we have assumed a fairly simple task model, and shown how FPS and EDF schedulability analysis can be applied. Extending the model to consider precedence constraints, blocking caused by shared resources, etc. should be straightforward, but the details need to be formalised.

The future work also includes investigating alternative ways to specify triggering patterns, for example timed automata [2] or RTL [17, 21]. The proposed algebra provides an intuitive high-level formalism for event pattern specification, but for some applications a more detailed control might be required.

References

[1] AbsInt company homepage. Accessed Mar 20, 2007. `http://www.absint.com`.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science. Springer, 2003.

[4] M. Berndtsson and J. Hansson. Issues in active real-time databases. In *Active and Real-Time Database Systems*, pages 142–157, 1995.

[5] Bound-T homepage. Accessed Mar 20, 2007. `http://www.tidorum.fi/bound-t/`.

[6] J. Carlson. *Event Pattern Detection for Embedded Systems.* PhD thesis, Mälardalen University, June 2007.

[7] J. Carlson and B. Lisper. An Event Detection Algebra for Reactive Systems. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT)*, 2004.

[8] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *IEEE Design Automation & Test in Europe (DATE)*, March 2003.

[9] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[10] A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. of Database and Expert Systems Applications 13th Int. Conference (DEXA'02)*, volume 2453 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 2002.

[11] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Edinburgh, UK, Sept. 1993. Springer-Verlag.

[12] N. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE: A system for composite specification and detection. In

Advanced Database Systems, volume 759 of *Lecture Notes in Computer Science*. Springer, 1993.

[13] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - the SymTA/S Approach. In *IEE Proceedings Computers and Digital Techniques*, 2005.

[14] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[15] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. 11^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 201–212, December 1990.

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[17] G. Liu, A. Mok, and P. Konana. A unified approach for specifying timing constraints and composite events in active real-time database systems. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 199–209, Washington - Brussels - Tokyo, June 1998. IEEE.

[18] G. Liu, A. K. Mok, and E. J. Yang. Composite events for network event correlation. In *Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management*, pages 247–260. IEEE, 1999.

[19] J. Mellin. *Resource-Predictable and Efficient Monitoring of Events*. PhD thesis, Department of Computer and Information Science, Linköping University, June 2004. Dissertation No 876.

[20] A. Mok and G. Liu. Early detection of timing constraint violation at runtime. In *The 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 176–186, Washington - Brussels - Tokyo, Dec. 1997. IEEE.

[21] A. K. Mok, P. Konana, G. Liu, C.-G. Lee, and H. Woo. Specifying timing constraints and composite events: An application in the design of electronic brokerages. *IEEE Transactions on Software Engineering*, 30(12):841–858, 2004.

[22] I. Motakis and C. Zaniolo. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3/4):291–325, 1997.

[23] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, 1996.

[24] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2/3):101–155, 2004.

[25] L. Sha, R. Rajkumar, and J. Lehoczky. Task scheduling in distributed real-time systems. In *IEEE Industrial Electronics Conference*, 1987.

[26] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, Institut National de Recherche et Informatique et en Automatique, 1996.

[27] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[28] Worst case execution times (WCET) project homepage. Accessed Mar 20, 2007. `http://www.mrtc.mdh.se/projects/wcet`.