End-to-End Latency Analyzer for ProCom - EELAP

Release 0.3.0

Jiří Kunčar <jiri.kuncar@gmail.com>
Rafia Inam <rafia.inam@mdh.se>
Mikael Sjödin <mikael.sjodin@mdh.se>

March 05, 2013

CONTENTS

1 Introduction **3**
- 1.1 End-to-End Delay Analysis . . . 3
- 1.2 Communication Strategies among ProCom Components . . . 3
- 1.3 Contributions . . . 4

2 User's Guide **5**
- 2.1 Installation . . . 5
- 2.2 Quick start . . . 5

3 API Reference **7**
- 3.1 API . . . 7

4 Additional Notes **19**
- 4.1 Licence . . . 19

Bibliography **21**

Index **23**

Abstract

This report presents an analysis tool *End-to-End Latency Analyzer for ProCom (EELAP)* developed to compute different end-to-end latency semantics for multi-rate components of real-time embedded systems. ProCom component technology implements executable reusable real-time components called *Runnalbe Virtual Nodes (RVNs)* and supports two different communication strategies for inter-RVN communication. The tool is developed to evaluate the two communication strategies for multi-rate server-based components the ProCom component technology.

In this report we present a user guide for EELAP. We describe the formulas and corresponding algorithms used to compute end-to-end latency semantics, and response-times of the tasks executing in a two-level hierarchical scheduling framework. Further, we provide a detailed description of API's.

CHAPTER

ONE

INTRODUCTION

Here we present a brief introduction to the terminologies required to understand the tool. We first explain end-to-end delay analysis and then describe two communication strategies among the ProCom components.

1.1 End-to-End Delay Analysis

In embedded systems, the realization of a piece of functionality can follow a flow through many software components. Data may originate at one component (e.g. a sensor) and passes through various other computational components, before terminating at the final component (e.g.an actuator). Hence, the data follows a chain of components $(C1, C2, \ldots, Cn)$, each potentially having its own periodicity and timing properties. The total time taken by the data/signal to traverse the complete chain is called *end-to-end latency* [2]. For an embedded system with real-time constraints, the end-to-end timing behavior is not only dependent on the timing properties of its constituent components but also on the message-chains among those components. In a communication chain, different executable components (or tasks) are activated at different periods. Such system is called a *multi-rate system* [2].

1.2 Communication Strategies among ProCom Components

In ProCom component model, the RVNs are implemented as servers (called RVN-server) and the tasks are executed within the servers. This type of systems are called server-based systems or hierarchical systems [3]. The RVN-servers are executed within a two-level hierarchical scheduling implementation [4]. To support communication *between* RVNs (also called *inter-RVN communication*), two different strategies have been proposed [5]. The first strategy is called a *server-based communication*, implemented using a *communication server*. The communication code is embedded within the communication server, which is activated periodically.

Another interesting approach is to use a *direct communication strategy*, where RVNs communicate with each other directly without an intermediate server. Here the communication code is encapsulated within the RVN to send and receive the data and/or messages. The details of both communication strategies for ProCom component model can be found in [5].

Both communication strategies for the ProCom technology reveal the multi-rate systems. According to [2], all automotive embedded systems are multi-rate systems. A system comprising the communication chains among RVN servers transposes to a multi-rate server-based system. Four different end-to-end semantics are provided in [2] for multi-rate systems. We develop the EELAP tool to compute these semantics for multi-rate server-based systems using (1) response times of tasks executed within a server and (2) then end-to-end semantics.

1.3 Contributions

We implement a tool *End-to-End Latency Analyzer for ProCom (EELAP)* [6] to evaluate timing behavior of two communication strategies in a multi-rate server-based real-time embedded components using end-to-end latencies (or delays). The tool computes end-to-end latencies using the following two steps:

- First it calculates the response times of all tasks executing in a two-level hierarchical scheduling framework by using methods/formulas provided in [1],
- And then it calculates different end-to-end latency semantics for both communication strategies.

In this report, we present the descriptions of API's of EELAP tool, the formulas and their implementations in those API, and the algorithms for schedulability condition, possible paths, path reachability, and generate paths for different latency semantics.

CHAPTER

TWO

USER'S GUIDE

2.1 Installation

This tool depends on two external libraries: the numpy and the argparse. These libraries are not documented here. If you want to dive into their documentation, check out the following links:

- Numpy Documentation
- Argparse Documentation

Besides numpy and argparse, the *lxml* is recommended for full functionality. *pip* or *easy_install* will install them for you if you do *pip install git+git://github.com/jirikuncar/eelap.git*. We encourage you to use a virtualenv.

2.2 Quick start

Eager to get started? This page gives a good introduction how the End-to-End Latency Analysis for ProCom works and how you can benefit from it. It assumes you already have it installed. If you do not, head over to the *Installation* section.

2.2.1 Finding possible execution paths

The whole simulation is dependant on quick and effective algorithm for finding possible execution paths of tasks in all system components. All latency types are calculated on specified data flow path that contains identifiers of tasks in analyzed system.

Our generator `generate_paths()` returns tuples with activation indexes of tasks accordingly to the analyzed execution path. The algorithm starts with finding closest activation indexes of the first task in path for defined interval. Following pseudo-code shows simplified version of our algorithm using methods `alpha()` and `ialpha()` defined on `Task`.

```
1  function generate_paths(start, stop, tasks_in_path):
2      paths <- list()
3      task <- tasks_in_path.pop(0)  # assign and remove first task from the path
4      loop i from task.ialpha(start) to task.ialpha(stop):
```

```
5           # find a time range for next task
6           if length(tasks_in_path) > 0:
7               time <- task.alpha(i)
8               next_task <- tasks_in_path[0]  # next task in path
9               j <- next_task.ialpha(time)  # closest activation index
10              new_start <- next_tasks.alpha(j)  # closest activation time
11              # find possible paths for next task in path from new start.
12              for all path in generate_paths(new_start, stop, tasks_in_path):
13                  # join current activation index with found tuple
14                  paths.append( path.prepend(i) )
15              end for
16          else:
17              paths.append( list(i) )  # list with only one index
18          end if
19      end loop
20      return paths
21  end function
```

CHAPTER

THREE

API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

3.1 API

This part of the documentation covers all the interfaces of End-to-End Analyzer for ProCom.

3.1.1 Task

class `eelap.`**`Task`**(*name*, *period*, *priority*, *exetime*)

System task.

`C`

Alias for task t_i execution time (C_i).

`HB`

Set of tasks belonging to same `Component` *C* with priorities higher than itself one.

$$HB(s) = \{T_k \in C \mid P(k) > P(s)\}$$

`P`

Alias for task t_i period ($P(i)$).

static `alpha`(i)

Calculate time of *i*-th activation.

$$\alpha_r(i) = r_{start} + i * P(r)$$

Parameters **i** – Activation number.

Returns Activation time.

`blocking_time`

Task deadline (b_i).

> **Warning:** Currently always returns 0.

deadline
Task deadline.

Note: Currently *deadline==period*.

---

static delta (*i*)
Calculate response time of *i*-th activation.

Parameters **i** – Activation number.

Returns Response time of *i*-th activation.

Note: In our case we always return task response time.

See Also:

`response_time()`

freq
Task frequency $f(t_i) = 1/P(t_i)$.

static ialpha (*t*)
Calculate previous activation for given time.

$$\alpha_r^{-1}(t) = \lfloor \frac{t - r_{start}}{P(r)} \rfloor$$

Parameters **t** – Current time.

Returns int – activation number.

p
Alias for task t_i priority ($p(i)$).

plan (*time*)
Returns True if the task should be scheduled at given *time*.

static rbf (*t*)
The request bound function.

It computes the maximum cumulative execution requests that could be generated from the time that task is released up to time *t*.

$$rbf_i(t) = C_i + b_i + \sum_{\tau_k \in HP(i)} \frac{t}{T_k} * C_k$$

Parameters **t** – Time limit for execution requests.

See Also:

[1] formula (2).

response_time
Task response time.

$$RS(i) = t \mid \exists t : 0 < t <= D_i, rbf_i(t) <= sbf_C(t)$$

status (*time*)
Returns textual representation of task status at given *time*.

utilization
Calculates utilization.

3.1.2 Component (Subsystem)

class eelap.**Component** (*name*, *period*, *priority*, *budget*, *scheduler='EDF'*, *payback=False*)
System servers / components.

Bl
The maximum blocking imposed to a this subsystem.

$$Bl(s) = max\{X(k) \mid S_k \in LPS(s)\}$$

See Also:

[1] formula (10).

HPS
Set of subsystems of System *S* with priority higher than itself (S_s).

$$HSP(s) = \{S_k \in S \mid P(k) > P(s)\}$$

See Also:

Used in formula (9) in [1].

LPS
Set of subsystems of System *S* with priority lower than itself (S_s).

$$LSP(s) = \{S_k \in S \mid P(k) < P(s)\}$$

See Also:

Used in formula (10) in [1].

P
Alias for Component period.

Q
Alias for Component budget.

RBF (*t*)
Request Bound Function

See Also:

[1] formula (9) and (11).

X
The maximum execution-time that any subsystem-internal task may lock a shared global resource.

> **Warning:** Currently always returns 0.

deadline
The function returns `Component` deadline.

Currently the *self.deadline==self.period*.

f1 (*t*)
Implementation of helper formula *f1*.

Parameters **t** – time

See Also:

[1] formula (5).

f2 (*t*)
Implementation of helper formula *f2*.

Parameters **t** – time

See Also:

[1] formula (6).

freq
The function calculates `Component` frequency.

static **sbf** (*t*)
Supply Bound Function.

If *payback* is *True* then this method depends on `f1()` and `f2()`:

```
return max(min(f1(t), f2(t)), 0)
```

Parameters **t** – time

See Also:

[1] formula (3) and (4).

schedulability
`Component` schedulability condition.

```
for t in C_{tasks}:
    schedulable <- False
    for i in [1..P(t)]:
        if rbf_t(i) <= sbf_C(i):
            schedulable <- True
            break
        end if
    end for
    if not schedulable:
        return False
    end if
end for
return True
```

See Also:

[1] formula (13).

utilization
Returns `Component` utilization as product of its frequency and budget.

3.1.3 System

class `eelap.`**System**(*scheduler='FPS'*, *resolution=1000*, *components=None*)
Models a physical system with instances of `Component`.

B(*t*)
Blocking time left at given *time*.

> **Warning:** Currently always returns 0.

static TP_first(*possible_paths*)
Set of all non-duplicate, reachable timed paths, for which no timed path exists that shares the same start instance of the first task and has an earlier end instance of the last task.

$$\mathbb{TP}^{first} = \{\vec{tp} \in \mathbb{TP}^{reach} \mid \neg\exists \vec{tp}' \in \mathbb{TP}^{reach} : tp'_1 = tp_1 \wedge tp'_n < tp_n\}$$

```
function earlier(tp, tp'):
    # index -1 gets last element in array
    return tp'[0] == tp[0] and tp'[-1] < tp[-1]
end function

out <- list()
TP_reach_paths <- TP_reach(possible_paths)
for tp in TP_reach_paths:
    if not any(map(partial(earlier, tp), TP_reach_paths)):
        out.append(tp)
```

```
    end if
end for
return out
```

See Also:

[2] formula (11).

static `TP_reach` (*possible_paths*)
It obtains the set of all paths and returns only all reachable timed path ($\mathbb{TP}^{reach}$).

```
out <- list()
for all path in possible_paths:
    if reach_path(path):
        out.append(path)
    end if
end for
return out
```

See Also:

[2] formula (9).

`addComponent` (*component*)
Adds new `Component` instance to the system.

It stores reference of the system to added component.

Parameters **component** (`Component`) – New system subsystem.

static `att` (*w*, *i*, *r*, *j*)
It returns *True* if "activation time travel" occurs ($att(t_w(i) \rightarrow t_r(j))$).

The activation time travel occurs when the reader is activated before the writer ($\alpha_r(i)$ is equivalent to `alpha()` on `t()`).

$$att(t_w(i) \rightarrow t_r(j)) = \alpha_r(j) < \alpha_w(i)$$

Parameters

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

See Also:

[2] formula (3)

static `crit` (*w*, *i*, *r*, *j*)
The "critical function" determines if writer and reader overlap in execution even in

case of non-activation time travel ($crit(t_w(i) \rightarrow t_r(j))$).

$$crit(t_w(i) \rightarrow t_r(j)) = \alpha_r(j) < \alpha_w(i) + \delta_w(i)$$

Parameters

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

See Also:

[2] formula (4)

delta_FF (*ls*)
Find maximum of First-to-First path delays.

See Also:

[2] formula (17).

delta_FF_path (*path*, *tp_reach*)
Calculate First-to-First path delay.

$$\Delta^{FF}(\vec{tp}) = \Delta^{LF}(\vec{tp}) + \alpha_1(tp_1) - \alpha_1(pred(\vec{(tp)}))$$

Parameters

- **path** – Array with task activation numbers.
- **tp_reach** – List of reachable paths.

Fixme remove dependency on *tp_reach* parameter.

See Also:

[2] formula (16).

delta_FL (*ls*)
Find maximum of First-to-Last path delays.

See Also:

[2] formula (15).

delta_FL_path (*path*, *tp_reach*)
Calculate First-to-Last path delay (uses `pred()`).

$$\Delta^{FL}(\vec{tp}) = \Delta^{LL}(\vec{tp}) + \alpha_1(tp_1) - \alpha_1(pred(\vec{(tp)}))$$

Parameters

- **path** – Array with task activation numbers.
- **tp_reach** – List of reachable paths.

Fixme remove dependency on *tp_reach* parameter.

See Also:

[2] formula (14).

static `delta_LF` (*ls*)

The maximum "Last-to-First" timed path delay.

$$\Delta^{LF}(p) = max\{\Delta(\vec{(}tp)) \mid \vec{(}tp) \in \mathbb{TP}^{first}\}$$

See Also:

[2] formula (12).

static `delta_LL` (*possible_paths*)

Returns maximum latency over all reachable paths (`TP_reach()`).

$$\Delta^{LL}(possible_paths) = max\{\Delta(path) \mid path \in \mathbb{TP}^{reach}\}$$

```
# map .. calls function for each element in list
# max .. returns maximal element from list
return max(map(delta_path, TP_reach(possible_paths)))
```

See Also:

[2] formula (10).

`delta_LL_path` (*path*)

Calculate Last-to-Last *path* delay using `delta_path()`.

`delta_path` (*path*)

Calculate end-to-end *path* delay.

$$\Delta(path) = \alpha_n(path_n) + \delta_n(path_n) - \alpha_1(path_1)$$

See Also:

[2] formula (2).

static `forw` (*w*, *i*, *r*, *j*)

It determines the forward reachability of the two task instances t_w and t_r.

$$forw(t_w(i) \to t_r(j)) = \neg att(t_w(i) \to t_r(j)) \wedge (\neg crit(t_w(i) \to t_r(j)) \vee wait(t_w(i) \to t_r(j)))$$

Parameters

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

See Also:

[2] formula (6)

generate_paths (*index*, *start*, *stop*)

Generator of possible paths for given task in path.

It yields tuples with task activation index starting from ‘index‘th task in defined path.

```
paths <- list()
task <- tasks_in_path.pop(0)  # assign and remove first task from the pa
loop i from task.ialpha(start) to task.ialpha(stop):
    # find a time range for next task
    if length(tasks_in_path) > 0:
        time <- task.alpha(i)
        next_task <- tasks_in_path[0]  # next task in path
        j <- next_task.ialpha(time)  # closest activation index
        new_start <- next_tasks.alpha(j)  # closest activation time
        # find possible paths for next task in path from new start.
        for all path in generate_paths(new_start, stop, tasks_in_path):
            # join current activation index with found tuple
            paths.append( path.prepend(i) )
        end for
    else:
        paths.append( list(i) )  # list with only one index
    end if
end loop
return paths
```

pred (*path*, *tp_reach*)

Temporal distance to the start of the latest previous “last-to-x” path.

See Also:

[2] formula (13).

static reach (*w*, *i*, *r*, *j*)

The output of an instance $t_w(i)$ is overwritten by instance $t_w(i+1)$ when both instances can forward reach the same reading task instance $t_r(j)$. In other words, $t_w(i)$ can reach $t_r(j)$ if and only if the following function returns *True*:

$$reach(t_w(i) \rightarrow t_r(j)) = (forw(t_w(i) -> t_r(j)) \wedge \neg forw(t_w(i+1) -> t_r(j)))$$

Parameters

- **w** – Index of writer task in data path.

- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

See Also:

[2] formula (7).

static reach_path (*path*)
Check *path* reachability.

```
path_length <- length(path)
for i in [0..path_length-1):
    tp_i  <- path[i]
    tp_i1 <- path[i+1]
    if reach(t_w(tpi) -> t_{w+1}(tp_i1)):
        return False
    end if
end for
return True
```

See Also:

[2] formula (8).

schedulability
This method checks the global schedulability condition.

```
for C in components:
    # P(C) .. period of component C
    schedulable <- False
    for t in [0..P(C)]:
        if RBF(C, t) <= t:
            schedulable <- True
            break
        end if
    end for
    if not schedulable:
        return False
    end if
end for
return True
```

See Also:

[1] formula (8).

t (*i*)
Get *i*-th `Task` instance (t_i).

Parameters **i** – The index of system task starting from 0.

Returns Instance of `Task`.

tasks
All system tasks.

tasks_in_path
List of tasks in data path.

utilization
Returns system utilization calculated as sum of component utilizations.

static wait (*w*, *i*, *r*, *j*)
It determines if the writer finishes first, because the reader has to wait due to its priority in case of overlapped but not time-traveling execution ($wait(t_w(i) \rightarrow t_r(j))$).

$$wait(t_w(i) \rightarrow t_r(j)) = p(t_r) < p(t_w)$$

Parameters

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

See Also:

[2] formula (5)

CHAPTER

FOUR

ADDITIONAL NOTES

Design notes, legal information and changelog are here for the interested.

4.1 Licence

Source code is distributed under following GNU/GPLv2 licence.

```
End-to-End Latency Analyzer for ProCom (EELAP) is free software;
you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the Free Software Foundation; either version 2 of the
License, or (at your option) any later version.

EELAP is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with Invenio; if not, write to the Free Software Foundation, Inc.,
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.
```

BIBLIOGRAPHY

[1] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, 51–60. Porto, Portugal, July 2011.

[2] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'08)*. 2008.

[3] R. Inam, J. Mäki-Turja, M. Sjödin, and J. Kuncar. Real-Time Component Integration using Runnable Virtual Nodes. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 12)*. Izmir, Turkey, September 2012. IEEE Computer Society.

[4] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei,, and S. Afshar. Support for Hierarchical Scheduling in FreeRTOS. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*. Tolouse, France, September 2011. IEEE Computer Society.

[5] R. Inam and M. Sjödin. Implementing and Evaluating Communication- Strategies in the ProCom Component Technology. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012), WiP*. ACM SIGBED Review, July 2012.

[6] J. Kunčar. End-to-End Latency Analyzer for ProCom (EELAP), https://github.com/jirikuncar/eelap/. 2013.

INDEX

addComponent() (eelap.System method), 12
alpha() (eelap.Task static method), 7
att() (eelap.System static method), 12

B() (eelap.System method), 11
Bl (eelap.Component attribute), 9
blocking_time (eelap.Task attribute), 7

C (eelap.Task attribute), 7
Component (class in eelap), 9
crit() (eelap.System static method), 12

deadline (eelap.Component attribute), 10
deadline (eelap.Task attribute), 8
delta() (eelap.Task static method), 8
delta_FF() (eelap.System method), 13
delta_FF_path() (eelap.System method), 13
delta_FL() (eelap.System method), 13
delta_FL_path() (eelap.System method), 13
delta_LF() (eelap.System static method), 14
delta_LL() (eelap.System static method), 14
delta_LL_path() (eelap.System method), 14
delta_path() (eelap.System method), 14

f1() (eelap.Component method), 10
f2() (eelap.Component method), 10
forw() (eelap.System static method), 14
freq (eelap.Component attribute), 10
freq (eelap.Task attribute), 8

generate_paths() (eelap.System method), 15

HB (eelap.Task attribute), 7
HPS (eelap.Component attribute), 9

ialpha() (eelap.Task static method), 8

LPS (eelap.Component attribute), 9

P (eelap.Component attribute), 9
P (eelap.Task attribute), 7
p (eelap.Task attribute), 8
plan() (eelap.Task method), 8
pred() (eelap.System method), 15

Q (eelap.Component attribute), 9

RBF() (eelap.Component method), 10
rbf() (eelap.Task static method), 8
reach() (eelap.System static method), 15
reach_path() (eelap.System static method), 16
response_time (eelap.Task attribute), 8

sbf() (eelap.Component static method), 10
schedulability (eelap.Component attribute), 10
schedulability (eelap.System attribute), 16
status() (eelap.Task method), 9
System (class in eelap), 11

t() (eelap.System method), 16
Task (class in eelap), 7
tasks (eelap.System attribute), 16
tasks_in_path (eelap.System attribute), 17
TP_first() (eelap.System static method), 11
TP_reach() (eelap.System static method), 12

utilization (eelap.Component attribute), 11
utilization (eelap.System attribute), 17
utilization (eelap.Task attribute), 9

wait() (eelap.System static method), 17

X (eelap.Component attribute), 10