LSMC-TN #02: SchedMCore task file format

Wolfgang Puffitsch, Alessandra Melani

October 23, 2015

Revision history

Version	Date	Comments
Version 1.0	February 19, 2013	First version
Version 1.1	March 6, 2013	Add user function support
Version 1.3	October 23, 2015	More detailed examples and grammar comments

1 Introduction

This note presents the task file format used by the SchedMCore[1] toolset and its evolution. The objectives of such a file format are:

- to describe each task with its real-time properties (deadline, WCET, release date etc. . .)
- to describe the dependencies between the tasks
- to describe the way tasks exchange data (through real of virtual buffers)
-

The first version of the SchedMCore Task file format (a.k.a. `TFF`) lacks support for the description of buffers between tasks and the mapping of tasks to processor cores. Furthermore, its grammar contains a shift/reduce conflict. Version 2.0 of this format seeks to overcome these limitations and to increase the readability of the descriptions. The source code of the grammar as a couple of lex and yacc specification is readily available in the schedmcore source code[2].

1.1 Disambiguation

The new file format must start with the token `TFF-2.0`, which is invalid in the old format. Therefore, parsers that understand only the old format will refuse to parse the new format and vice versa. Parsers that support both formats can use this token for disambiguation. Further evolution of the file format should keep this way of maintaining disambiguation.

[1] http://sites.onera.fr/schedmcore/

[2] Lex file:`https://svn.onera.fr/schedmcore/trunk/lib/lsmc_taskfile_tokens.ll`, Yacc file:`https://svn.onera.fr/schedmcore/trunk/lib/lsmc_taskfile_syntax.yy`

1.2 Changes

The new format adds support for the description of buffers and mappings of tasks to processor cores. Furthermore, it allows the specification of more general dependencies and deadline patterns.

A delicate change is that in the original format, the description of tasks contained the task properties in the order <period, WCET, deadline, release date>. The new format switches the order of the release date and the deadline and places the description of the deadline at the end. This change was made to retain readability even in case of long deadline patterns.

1.3 Restrictions

The new format allows the expression of precedences with a prefix and sequences of deadlines with a prefix and a repeating pattern. Support for these features is not mandatory, as expressed by the grammar in Section 4. This grammar is a strict subset of the grammar described in Section 3, and the grammar to be supported by SchedMCore and Interlude. Task descriptions generated by Prelude or Interlude use the full grammar.

2 Taskfile Format v1.0 Grammar

```
1  tasks_specs : spec_list
2
3  spec_list : /* empty */
4          | spec_list spec_line
5
6  spec_line : task_spec_line
7          | depend_spec_line
8          | user_function_spec_line
9
10 task_spec_line : 'Task' string int int int int
11         | int int int int /* period WCET deadline release date */
12
13 depend_spec_line : 'Dependency' string string dword_spec
14
15 dword_spec : int int
16         | dword_spec int int
17
18 user_function_spec_line : 'UserFunction' string 'in' string 'for' string
19
```

At the lines 10 and 11 of the grammar the integer quadruplet of this v1 (first version) file format are: period, WCET deadline and release date in that order.

3 Taskfile Format v2.0 Grammar

```
tasks_specs : 'TFF-2.0' spec_list

spec_list : /* empty */
        | spec_list spec_line

spec_line : task_spec_line
        | depend_spec_line
        | combuffer_spec_line
        | map_spec_line
        | user_function_spec_line

    /* Task Name [:= Userfunction] Period WCET Offset Deadlines */
task_spec_line : 'Task' string int int int deadline_spec
        | 'Task' string ':=' string int int int deadline_spec

deadline_spec : deadline_prefix '(' deadline_pattern ')'

deadline_prefix : /* empty */
        | deadline_prefix int ','

deadline_pattern : int
        | deadline_pattern ',' int

    /* Dependency From To Pattern */
depend_spec_line : 'Dependency' string string dword_spec

dword_spec : dword_prefix '(' dword_pattern ')'

dword_prefix : /* empty */
        | dword_prefix dword ','

dword_pattern : dword
        | dword_pattern ',' dword

dword : int ':' int

    /* ComBuffer From To ElementSize NbElements [:= InitFunction] */
combuffer_spec_line : 'ComBuffer' string string intvalue intvalue
        | 'ComBuffer' string string intvalue intvalue ':=' string

    /* Map Task Core */
map_spec_line : 'Map' string intvalue

    /* UserFunction Function in File */
user_function_spec_line : 'UserFunction' string 'in' string
```

4 Taskfile Format v2.0 Restricted Grammar

```
tasks_specs : 'TFF-2.0' spec_list

spec_list : /* empty */
        | spec_list spec_line

spec_line : task_spec_line
        | depend_spec_line
        | combuffer_spec_line
        | map_spec_line
        | user_function_spec_line

   /* Task Name [:= Userfunction] Period WCET Offset ( Deadline ) */
task_spec_line : 'Task' string int int int deadline_spec
        | 'Task' string ':=' string int int int deadline_spec

deadline_spec : '(' int ')'

   /* Dependency From To Pattern */
depend_spec_line : 'Dependency' string string dword_spec

dword_spec : '(' dword_pattern ')'

dword_pattern : dword
        | dword_pattern ',' dword

dword : int ':' int

   /* ComBuffer From To ElementSize NbElements [:= InitFunction] */
combuffer_spec_line : 'ComBuffer' string string intvalue intvalue
        | 'ComBuffer' string string intvalue intvalue ':=' string

   /* Map Task Core */
map_spec_line : 'Map' string intvalue

   /* UserFunction Function in File */
user_function_spec_line : 'UserFunction' string 'in' string
```

5 Examples

5.1 Example v1.0

```
Task "task 1" 6 1 6 0
Task "task 2" 4 1 4 3
Dependency "task 2" "task 1" 0 0
```

5.2 Example v2.0

```
TFF-2.0
Task "task 1" 6 1 0 (6)
Task "task 2" 4 1 3 (4)
Dependency "task 2" "task 1" (0:0)
```

5.3 Example v2.0 Full Grammar

```
Task "i0"       10    1   0    7,(7,7)
Task "o0"       5     1   0    (5)
Task "n8"       10    2   0    9,(4,4)
Dependency "i0" "n8"  (0:0)
Dependency "n8" "o0"  0:1,(0:1,1:2)
ComBuffer "n8.o"    "o0.o"   4  2 := "init_n8_o0"
ComBuffer "i0.i"    "n8.i"   4  1 := "init_i0_n8"
Map "i0" 0
Map "o0" 1
```

5.4 Example v2.0 Restricted Grammar

```
Task "i0"       10      1       0       (7)
Task "o0"       5       1       0       (5)
Task "n8"       10      2       0       (4)
Dependency "i0" "n8"    (0:0)
Dependency "n8" "o0"    (0:1,1:2)
ComBuffer "n8.o"        "o0.o"  4       2       := "init_n8_o0"
ComBuffer "i0.i"        "n8.i"  4       1       := "init_i0_n8"
Map "i0" 0
Map "o0" 1
```

6 User functions

The current task file format (both versions 1.0 and 2.0) supports the inclusion of user defined functions, specified as C functions, which can dynamically be linked to tasks and executed at every job instance. C functions must be contained inside a dynamic library (under most Unix, extension .so), whose location is retrieved through the path specified by the environment variable `LD_LIBRARY_PATH`.

For the time being, parameters cannot be specified inside the text file, since the grammar requires only to specify the name of the function, the library which contains it and the task which is in charge of executing the function. Nevertheless, since most of the functions we may need do have input arguments, it is necessary to build wrapper functions with no arguments which invoke the real functions, specifying the inputs and possibly making them vary.

A C file (together with its header file) is included in the library we may want to use, for example:

```
#include <stdio.h>
#include "usr.h"

int
userFunTest(void)
{
  //do something
  printf("UserFunction terminated...\n");
}
```

6.1 Examples

In the following, we specify how this function could be invoked exploiting text files v1.0 and v2.0.

6.1.1 Example v1.0

```
Task "Lg" 40 4 40 0
Task "Gg" 40 4 40 0
Task "Lp" 20 3 20 0
Task "Fp" 20 3 20 0
Task "As" 10 1 10 0
Task "Fa" 10 1 10 0
Task "Ap" 10 1 10 0
Dependency "Fp" "Gg" 0 0
Dependency "Ap" "As" 0 0
Dependency "Fp" "Lp" 0 0
UserFunction "userFunTest" in "libUserFun.so" for "As"
```

6.1.2 Example v2.0

```
TFF-2.0
Task "Lg" 40 4 0 (40)
Task "Gg" 40 4 0 (40)
Task "Lp" 20 3 20 (20)
Task "Fp" 20 3 20 (20)
Task "As" 10 1 10 (10)
Task "Fa" 10 1 10 (10)
Task "Ap" 10 1 10 (10)
Dependency "Fp" "Gg" (0:0)
Dependency "Ap" "As" (0:0)
Dependency "Fp" "Lp" (0:0)
UserFunction "userFunTest" in "libUserFun.so" for "As"
```

In this example, the function `userFunTest.c`, belonging to the dynamic library `libUserFun.so`, is associated to task `As`. Hence, as soon as each job of task `As` is invoked, the function gets executed.