DSX tool specification

A) Goals and general principles

DSX stands for *Design Space eXplorer*. It helps the system designer to map a multi-threaded software application on a multi-processor hardware architecture (MP-SoC) modeled with the SoCLib components.

It supports the hardware software codesign approach, allowing the designer to define successively :

- the software application structure : number of tasks and communication channels
- the hardware architecture : number of processors, number of memory banks, etc.
- the mapping of the software application on the hardware architecture

A specific goal of DSX is to allow the system designer to control not only the placement of the tasks on the processors, but the placement of the software objects (execution stacks, communication buffers, synchronization locks, etc.) on the memory banks. In shared memory multi-processors architectures with several physically distributed memory banks, such control is mandatory to optimize both the performances and the power consumption.

The two targeted application domains are the telecommunication applications (where the tasks are handling packets or packet descriptors), and multi-media applications (where the tasks are handling audio or video streams).

The general principles of the DSX tool are the following:

- The coarse grain parallelism of the software application must be statically defined as a **Task & Communications Graph (TCG)**. The number of tasks, and the communication channels between tasks should not change during execution.
- The software tasks are supposed to be written in C or C++, but - for portability reasons - the tasks must use an abstract **System Resource Layer (SRL)** API to access the communication and synchronizations resources.
- Each task in the TCG can be implemented as a **software task** (software running on an embedded processor), or can be implemented as an **hardware task**, (running as a dedicated hardware coprocessor).
- DSX allows the programmer to use unprotected shared memory spaces, but the prefered inter-tasks communication mechanism use the **MWMR middleware**. The MWMR (Multi-Writer, Multi-Reader) channels, are implemented as software FIFOs and can be shared by *software tasks*, and by *hardware tasks*.
- DSX provides classical synchronization mechanisms such as barriers and locks, but inter-task synchronisation is mainly done through the data availability in the MWMR channels.

- The target hardware architecture is a shared memory multi-processor system on chip (MP-SoC) using the SoCLib library of IP cores. But - in order to validate the multi-threaded software application - DSX is able to generate an executable binary code for a standard POSIX workstation.
- DSX supports the POSIX compliant ?Mutek OS kernel for embedded MPSoCs
- Finally, DSX defines the DSX/L language, based on PYTHON, that allows the system designer to describe in a single file the Task & Communication Graph (TCG), the MP-SoC hardware architecture, and various mapping of the TCG on the MP-Soc architecture.

The DSX/L script execution generates the binary code executable on the workstation, the SystemC model of the *top cell* correspondint to the MP-SoC architecture, and the binary code that will be uploaded in the MP-Soc embedded memory.

B) System Resources Layer

We want to map the multi-threaded software application on several hardware platforms, without any modification of the task code. One important platform is a POSIX compliant workstation, as we want to validate the multi-threaded software application on a workstation before starting the mapping on the MPSoC architecture.

DSX defines a **system Ressource Layer** API (SRL), that is an abstraction of the synchronization and communication services provided by the various target platforms. The SRL API helps the C programmer to distinguish the embedded application code from the system code used for inter-tasks communications and synchronizations.

- Communications : blocking & non-blocking Read & Write access to a MWMR channel

```
void srl_mwmr_read( srl_mwmr_t, void * , size_t ) ;
void srl_mwmr_write( srl_mwmr_t, void * , size_t ) ;

size_t srl_mwmr_try_read( srl_mwmr_t, void * , size_t ) ;
size_t srl_mwmr_try_write( srl_mwmr_t, void * , size_t ) ;

void srl_mwmr_flush( srl_mwmr_t ) ;
```

- Synchronization barrier

```
void srl_barrier_wait( srl_barrier_t ) ;
```

- taking and releasing a lock

```
srl_loock_lock( srl_lock_t ) ;
srl_lock_unlock( srl_lock_t ) ;
```

- accessing a shared memory space (address and size)

```
void* srl_memspace_addr( srl_memspace_t ) ;
size_t srl_memspace_size( srl_memspace_t ) ;
```

Three platforms are presently supported :

- Any Linux (or Unix) workstation supporting the POSIX threads,
- MP-SoC architecture using the MUTEK/D operation system,
- MP-SoC architecture using the MUTEK/S operating system,

MUTEK/D is an embedded, POSIX compliant, distributed, operating system for MP-SoCs?, while MUTEK/S is an optimized version: the performances are improved, and the memory footprint is reduced, at the cost of loosing the POSIX compatibility.

C) Defining the software application

This chapter describes the DSX/L syntax used to define the Task & Communication Graph structure. The TCG is a bipartite graph: the two types of nodes are the tasks and the communication channels.

As an example, the following figure describes the TCG corresponding to an MJPEG decoder application. The two TG & RAMDAC tasks will be implemented as hardware coprocessors : the TG component implements a wire-less receiver for the MJPEG stream, and the RAMDAC component is a graphic display controller. The 5 other tasks can be implemented as *software tasks* or as *hardware tasks*. In this particular example, all MWMR communication channels have one single producer, and one single consumer, which is frequent for stream oriented multi-media applications.

C1) Task Model definition

As a software application can instanciate several instances of the same task, we must distinguish the task, and the task model. A task model defines the code associated to the task, and the task interface (corresponding to the system resources used by the task : MWMR communications channels, synchronization barriers, locks, and memspaces).

```
Task_Model = TaskModel( 'model_name',
                    infifos = [ 'inport_name', ... ] ,
                    outfifos = [ 'outport_name', ... ] ,
                    locks = [ 'lock_name', ... ] ,
                    barriers = [ 'barrier_name', ... ] ,
                    memspaces = [ 'memspace_name', ... ] ,
                    signals = [ 'signal_name', ... ] ,
                    impls = [ SwTask( 'func', stack_size = 1024 , sources = [ 'func.c' ] )
```

If a task does not use a given type of resource, the corresponding parameter can be skipped.

C2) MWMR communication channel definition

A MWMR communication channel is a memory buffer handled as a software FIFO that can have several producers and several consumers. Each channel is protected by an implicit lock for exclusive access. Any MWMR transaction can be decomposed in five memory access:

1. get the lock protecting the MWMR (READ access).
2. test the status of the MWMR (READ access).
3. transfer a burst of data between a local buffer and the MWMR (READ/WRITE access).
4. update the status of the MWMR (WRITE access).
5. release the lock (WRITE access).

Any data transfer to or from a MWMR channel mut be an integer number of items. The item width is an intrinsic property of the channel. It is defined as a number of bytes, and it defines the channel *width*. The channel *depth* is a number of items, and defines the total channel capacity. For performances reasons the channel *width* itself must be a multiple of 4 bytes.

```
My_Channel = Mwmr( 'channel_name', width, depth )
```

In the mapping section of the DSX/L program, the 4 following software objects must be placed :

1. *desc* : read only informations regarding the communication channel
2. *status* : channel state (number of stored items, read & write pointers)
3. *buffer* : channel buffer containing the data

4. *lock* : lock protecting exclusive access

C3) Synchronization barrier definition

The synchronization barriers can be used when the synchronization through the data availability in the MWMR communication channels in not enough. The set of tasks that are linked to a given barrier is defined when the the tasks are intanciated. Exclusive access to the barrier is protected by an implicit lock.

```
My_Barrier = Barrier( 'barrier_name' )
```

In the mapping section of the DSX/L program, the 3 following software objects must be placed :

1. *desc* : read only informations regarding the synchronization barrier
2. *status* : barrier state
3. *lock* : lock protecting exclusive access

C4) Memspace definition

Direct communication through shared memory buffers is supported by DSX, but there is no protection mechanism, and the synchronization is the programmer responsability. A shared memory space is defined by two parameters : *memspace_name* is the name, and *size* defines the number of bytes to be reserved.

```
My_Shared_Buffer = Memspave( ''memspace_name', size )
```

In the mapping section of the DSX/L program, the 2 following software objects must be placed :

1. *desc* : read only informations regarding the memspace
2. *mem* : the shared memory buffer

C5) lock definition

A lock is a variable that can be used to protect exclusive access to a shared resource such as a shared memory space. It is implemented as a spinlock : the *srl_lock_lock()* funtion returns only when the lock has been obtained.

```
My_Lock = Lock( 'lock_name' )
```

In the mapping section of the DSX/L program, the lock can be explicitely placed in the memory space.

C6) Signal definition

The DSX signals are used to signal a special event that is not synchronized with the data. The signal is transmitted to all registered tasks. The tasks are interrupted to execute the corresponding signal handler. Signals are mainlly used to implement soft real time constraints, a task can receive a signal, but cannot send a signal.

```
My_signal = Signal( 'signal_name' )
```

There is nothing to place in the mapping section.

C7) Task instanciation

A task is an instance of a task model. The constructor arguments are the task name *task_name*, the task model *Task_Model* (created by the TaskModel?() function), a list of resources (MWMR channels, synchronization barriers, locks or memspaces), and the list of the signals that can be received by the task . DSX performs type

checking between the port name and the associated resource.

```
My_Task = Task( 'task_name',
                                Task_Model ,
                               { 'port_name' : My_Channel, 'barrier_name' : My_Barrier, ... } ,
                               { 'signal_name : My_signal, ... } )
```

In the mapping section of the DSX/L program, 4 software objects must be placed :

1. *desc* : read-only informations associated to the task
2. *status* : state of the task
3. *stack* : execution stack
4. *run* : processor running the task

A task that has real time constraints must be instanciated by a special constructor. There is two extra arguments : *cond_activate* is the signal definig the activation condition, and *cond_deadline* is the signal defining the dead_line condition.

```
My_RT_Task = RtTask( 'task_name',
                                Task_Model ,
                               { 'port_name' : My_Channel, 'barrier_name' : My_Barrier, ... } ,
                               { 'signal_name : My_signal, ... } ,
                               cond_activate ,
                               cond_deadline )
```

D) Defining the hardware architecture

E) Mapping the software on the hardware

F) Code generation