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

D) Defining the hardware architecture

E) Mapping the software on the hardware

F) Code generation