DSX tool documentation

Authors : Nicolas Pouillon & Alain Greiner

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 also 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 multimedia applications (where the tasks are handling audio or video streams).

The general principles of DSX 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 regions, but the prefered inter-tasks communication mechanism use the **MWMR middleware**. The MWMR (Multi-Writer, Multi-Reader) communication 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. 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
- 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 simulator 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 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 , that is an abstraction of the synchronization and communication services provided by the various target platforms. The SrlApi helps the C programmer to distinguish the embedded application code from the system code used for inter-tasks communications and synchronizations.

- blocking Read & Write access to a MWMR channel
  - **srl_mwmr_read( )**
  - **srl_mwmr_write( )**
- non blocking Read & Write access to a MWMR channel
  - **srl_mwmr_try_read( )**
  - **srl_mwmr_try_write( )**
- flush a MWMR channel
  - **srl_mwmr_flush( )**

- Synchronization barrier
  - **srl_barrier_wait( )**

- taking and releasing a lock
  - **srl_lock_lock( )**
  - **srl_lock_unlock( )**

- accessing a shared memory space (address and size)
  - **srl_memspace_addr( )**
  - **srl_memspace_size( )**

Three platforms are currently supported :

- Any Linux (or Unix) workstation supporting the POSIX threads,

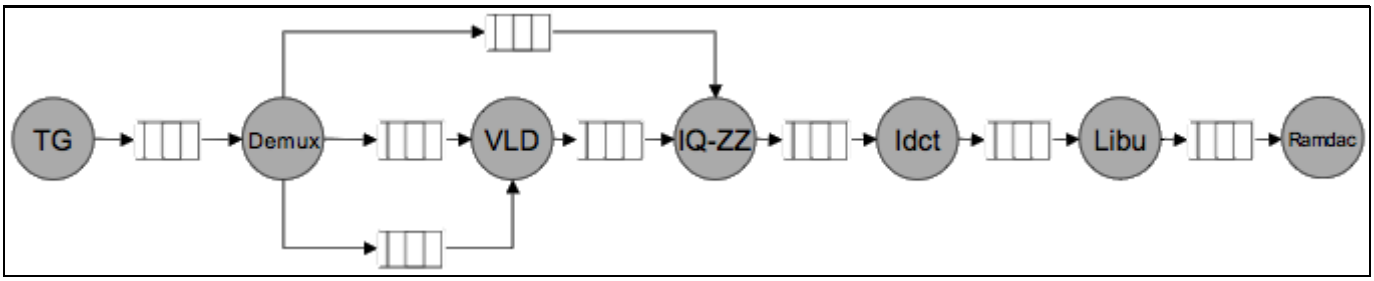
- MP-SoC architecture using the Mutek/S operating system (SRL directly implemented on Hexo),
- MP-SoC architecture using the Mutek/H operation system (SRL with Posix threads on Hexo),

Mutek/H 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) Software application definition

This section 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 wireless 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 multimedia applications.

C1) Task Model definition

As a software application can have 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, memspaces, ...).

```
TaskModel( 'model_name',
                    ports = { 'inport' : MwmrInput(32) ,
                              'ouport'  : MwmrOutput(64) ,
                              'my_barrier' : BarrierPort() ,
                              'my_buffer' : MemspacePort( 4096 ) }
                    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 (LL access).
2. get the lock protecting the MWMR (SC access).
3. test the status of the MWMR (3-words READ access).
4. transfer a burst of data between a local buffer and the MWMR (READ/WRITE access).

5. update the status of the MWMR and release the lock (4-words WRITE access).

Any data transfer to or from a MWMR channel mut be an integer number of items. The item is a number of bytes, and defines the channel *width*. The channel *depth* is a number of items, and defines the total channel capacity. For performances reasons the channel *width* 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

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_buffer = Memspace( 'buffer_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, 1 software object must be placed :

1. *lock* : Where to place the lock

C6) Task instanciation

A task is an instance of a task model. The constructor arguments are the task name *task_name*, the task model *model_name*, refered by name (created by the TaskModel?() function), and a list of resources (MWMR channels, synchronization barriers, locks or memspaces), that must be associated to the task ports. DSX performs type checking between the port name and the associated resource.

```
my_task = Task( 'task_name',
                         'model_name' ,
                         portmap = { 'port_name' : my_channel, 'barrier_name' : my_barrier, ...
```

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

C7) TCG definition

The Task and Communication Graph must be defined. The *portmap* construct is a list of assignments : the first argument is a port name. The second argument is the ressource, that can be a communication channel, a barrier, a lock, etc.

```
my_tcg = Tcg(
    Task(  'task_name1',   'model_name1',
                portmap = { ?in':channelA,
                                    'out':channelB } ) ,
    Task(  'task_name2',   'model_name2',
                portmap = { ?in':channelB,
                                    'out':channelC } ) ,
      ... )
```

D) Hardware architecture definition

This section describes the DSX/L syntax used to define the MP-SoC hardware architecture, using the hardware components defined in the SoCLib library.

D1) SoCLib components

The list of available components can be found in SoclibComponents. As it is possible to mix CABA and TLM-T simulation models in the same hardware architecture, the system designer can specify the type of simulation model used for each hardware component. For all components, the instance name is mandatory.

```
# definition of the hardware architecture
archi = soclib.Architecture()

# instanciation of a MIPS R3000 processor (MIPS ISS in a CABA wrapper), with processorID = 0
my_proc = archi.create('caba:iss_wrapper', 'proc', iss_t = 'common:mipsel', ident = 0)

# instanciation of a CABA cache controler. Both instruction and data caches contain 32 lines of
my_cache = archi.create('caba:vci_xcache', 'cache',
                        dcache_lines = 32,
                        dcache_words = 8,
                        dcache_lines = 32,
                        dcache_words = 8)
```

D2) Connecting the components

Hardware components have input/output ports, and are connected through signals, but those signals are implicit in the DSX/L description. To connect the port **a** of component **c1** to the port **b** of component **c2**, DSX/L define the *operator :*

```
c1.a // c2.b
```

Depending on the component type, the port designation can vary :

- For a single port : My_Proc0.dcache define the data cache port of the processor.
- For an array of ports : My_Proc0.irq[3] define the irq line 0 of the processor.
- When the number of port in a port array is not fixed (typically for interconnect component), the ports are allocated through the new() method on the port array.

The following example describes asimple system with two processor and on e embedded memory:

```
# components instanciacion
my_proc0 = archi.create('caba:iss_wrapper', 'proc0', iss_t = 'common:mipsel', ident = 0)
my_cache0 = archi.create('caba:vci_xcache', 'cache0',
                         dcache_lines = 32,
                         dcache_words = 8,
                         dcache_lines = 32,
                         dcache_words = 8)
my_proc1 = archi.create('caba:iss_wrapper', 'proc1', iss_t = 'common:mipsel', ident = 1)
my_cache1 = archi.create('caba:vci_xcache', 'cache1',
                         dcache_lines = 32,
                         dcache_words = 8,
                         dcache_lines = 32,
                         dcache_words = 8)
my_ram = archi.create('caba:vci_ram', 'ram' )
my_vgmn = archi.create('caba:vci_vgmn', 'vgmn' )

# components connexion
my_proc0.dcache // my_cache0.dcache
my_proc0.icache // my_cache0.icache
my_proc1.dcache // my_cache1.dcache
my_proc1.icache // my_cache1.icache
my_vgmn.from_initiator.new() // my_cache0.vci
my_vgmn.from_initiator.new() // my_cache1.vci
my_vgmn.to_target.new() // my_ram.vci
```

D3) Address space segmentation

In any shared memory architecture, the address space is a shared resource. This resource is structured in several segments. A segment has a name, a base address, a size (number of bytes), and a cacheability attribute (boolean). A segment is a physical entity associated to a given VCI target. Several segments can be associated to the same VCI target, but a given segment cannot be distributed over several VCI targets.

The DSX/L language allows the system designer to define the various segments used by a given application, and to link those segments to the hardware components.

```
my_ram = archi.create('caba:vci_ram', 'ram' )
my_ram.addSegment('cram0', 0x40000, 0x320)
my_ram.addSegment('mipsel_reset', 0xbfc00000, 0x100)
```

As a segment is defined by the MSB bits of the VCI address, the hardware interconnect must decode those MSB bits to select the proper VCI target. The corresponding decoder is generally implemented as a ROM. Those

hardware decoders are automatically constructed using the **Mapping Table**. The mapping table is an associative table. Each entry corresponds to a physical segment. This object must be constructed, and initialised:

```
# mapping table construction
my_mt = archi.create('common:mapping_table', 'mt', [4], [4], 0xc00000)
```

D4) Generic platforms

As DSX/L is based on Python, it is possible to define generic, parametrized architectutes, that can be reused for various applications. Those reusable architectures are functions.

```
#########################
# generic architecture definition

def MultiProc(nbcpu = 2):              # the argument value is the default value
archi = soclib.Architecture()

    vgmn = archi.create('caba:vci_vgmn', 'vgmn' )

    for i in range(nbcpu):
        proc = archi.create('caba:iss_wrapper', 'proc%d'%i, iss_t = 'common:mipsel', ident = i)
        cache = archi.create('caba:vci_xcache', 'cache%d'%i,
                        dcache_lines = 32,
                        dcache_words = 8,
                        dcache_lines = 32,
                        dcache_words = 8)

        proc.dcache // cache.dcache
        proc.icache // cache.icache

    my_ram = archi.create('caba:vci_ram', 'ram' )
    my_ram.addSegment( ?reset?, 0xbfc00000, true )
    my_ram.addSegment( ?code?, 0x40000000, true )
    my_ram.addSegment( ?data0?, 0x20000000, true )
    my_ram.addSegment( ?data1?, 0x30000000, false )

    my_vgmn.to_target.new() // my_ram.vci

#########################
# generic architecture instanciation

my_architecture = MultiProc( nbcpu = 4 )
```

E) Mapping the software on the hardware

At this point, we have defined the object **my_tcg** (defining the software application), and the object **my_architecture** (defining the hardware architecture). This section describes the DSX/L syntax used to map the TCG on the hardware architecture.

E1) Mapper declaration

As it is possible to define various mapping for a given TCG, and a given architecture, we must define a *mapper* object. This *mapper* will contain all the mapping directives defined by the system designer for a given mapping.

```
my_mapper = Mapper( my_architecture, my_tcg )
```

E2) Mapper definition

The mapper has a method `map()` that is used to assign a software object to an hardware component. An hardware component can be a processor, a segment associated to an embedded memory bank, or a segment associated to an addressable peripheral.

```
# Mapper definition
my_mapper = Mapper( my_architecture, my_tcg )

# For a MWMR channel, 4 software elements must be placed
my_mapper.map( my_channel,
        status = 'data0',  # The channel status is placed in segment data0
        desc = 'data1', # The channel descriptor is placed in segment data1
        buffer = 'data2' ) # The channel buffer is placed in segment data2

# for a software task, 4 software objects must be placed
my_mappe.map( my_task,
        desc = 'data0',  # The task descriptor is placed in segment data0
        status = 'data1',  # The task state is placed in segment data1
        stack = 'data2',   # The private task stack  is placed in segment data2
        run = 'proc0' )   # task will be running on processor proc0
```

F) Code generation

At each step in a DSX/L program, it is possible to generate output files. DSX defines various drivers corresponding to various outputs : binary code for the software application, hardware architecture simulation model, etc.

This involves a code generator. Several code generator exist, they may apply to different parts of you design:

- Software only (Tcg object)
  - ♦ Posix() for generating native workstation code
- Software and hardware (Mapper object). Those always generate the associated SystemC simulator
  - ♦ MutekS() to use Mutek/S as supporting embedded OS
  - ♦ MutekH() to use Mutel/H as supporting embedded OS
- Hardware only (Hardware object)
  - ♦ soclib.PfDriver?() to create a SystemC netlist (with SoCLib)

Example: Let's create

- An application mapped on an hardware platform with SystemC simulators
- a corresponding application for the workstation

```
soft = Tcg( ... )
hard = soclib.Architecture( ... )

mapping = Mapper( hard, soft )

mapping.map( ... )

# Generators now:

muteks_generator = MutekS()

posix_generator = Posix()

# MutekS and simulators (Caba / Tlmt) generates platform and embedded software for a mapping:

mapping.generate( muteks_generator )
```

```
# Posix generates code for a Tcg

tcg.generate( posix_generator )
```