A Model Based Safety Critical Flow for the AURIX Multi-core Platform

Bruno Pagano, Cédric Pasteur, Günther Siegel

A Model Based Safety Critical Flow for the AURIX™ Multi-core Platform

Bruno PAGANO, Cédric PASTEUR, Günther SIEGEL (ANSYS System Business Unit - France)

Roman KNÍŽEK (HighTec EDV-Systeme GmbH)

bruno.pagano@ansys.com cedric.pasteur@ansys.com gunther.siegel@ansys.com
roman.knizek@hightec-rt.com

Abstract

In this paper, we present a model based development flow for safety critical applications that targets parallel execution on Infineon's latest generation AURIX™ multi-core chips.

The flow is based on an extension of SCADE Suite®, that generates parallel code from any legacy Scade application. SCADE Suite is a product line of the ANSYS® Embedded software family of products and solutions that empowers users with a Model-Based Development Environment for critical embedded software. The generated parallel code is target independent; an integration step is performed to instantiate OS specific services like communication, synchronization or memory protection.

An automated integration process has been developed for the PXROS-HR operating system. PXROS-HR, developed by HighTec EDV-Systeme, is a multicore real-time operating system designed for embedded safety-critical applications. PXROS-HR is able to take full advantage of AURIX performance and safety functionalities.

SCADE Suite certified generated code embedded in PXROS-HR certified RTOS provide the streamlined answer to challenging development of critical multi-core embedded applications by enabling seamless automated flow from the application model down to executable binary running on a target embedded processor.

1. Overview

Many embedded systems are inspired by the benefits of multi-core CPUs. These benefits include the ability to deliver more performance with an acceptable power consumption and weight, the use of additional computation power to implement new functionalities, and finally anticipation of the mass market obsolescence for single-core processors.

But programming multi-core applications is a challenging task. Developers of embedded software are alone with complexity and are being forced to figure out ways to exploit multi-core platforms with languages and libraries generally not designed for multiprocessor environments. At the same time, they face increasingly stringent requirements on functional safety across the entire industrial and automotive application field.

Related work. Synchronous dataflow programming is naturally well-suited to parallel execution, thanks to the fact that all data dependencies and communications are explicit. There are thus no issues with concurrent accesses to shared memory or aliasing. Earlier work on synchronous languages has focused on the distribution of one application on several computation units (see for instance this survey [Girault]). In most cases, these solutions are based on a Globally Asynchronous Locally Synchronous (GALS) model, in which synchronous programs communicate via asynchronous message sending.

More recent work focuses on the scheduling and mapping of tasks to multi-core processors, with for instance the Synchronous Data Flow (SDF) [Lee & Al.] and Cyclo-Static Data Flow (CSDF) [Stuik & Al] models, or the mapping of Prelude programs to many-core architectures [Puffitsch & Al.]. These approaches are not adapted to the context of Scade because they only consider minimal languages using nodes imported from other languages (eg. Prelude is called an architecture language by its creators). In our case, we want to parallelize existing Scade 6 models which do not fit in these formalisms.

Another work proposes the addition of futures in a Scade-like language [Cohen & Al.]. However, this extension requires language changes and specific runtime support, which is not compatible with our objectives of compatibility and flexibility.

Contributions. At ANSYS, we have extended our code generator to support multi-core targets. The solution we have developed guarantees that the generated parallel code has a predictable deterministic behavior and that the sequential and parallel versions have the same behavior. The solution chosen to express the parallelism is

usable on existing models with minimal modifications. It allows parallelizing software execution without any modification of the Scade model, only light annotations.

The generated code is also target independent. The hardware and software architectures used in critical embedded systems differ widely from one domain to another. The code that is generated can easily be integrated into very different contexts such as the number of cores (from 2 to several hundred), the nature of the communications (by shared memory or by message passing) or the real-time operating system used (or its absence).

This code generator will be qualified/certified for various safety standards (DO-178C/DO-330 at TQL-1, IEC 61508 at SIL 3, EN 50128 at SIL 3/4, and ISO 26262 software up to ASIL D).

Certified code generation is only one part of the flow. It was mandatory to integrate this code on a certified RTOS.

PXROS-HR from HighTec is a multicore real-time operating system. Its micro-kernel provides encapsulation of individual tasks and communication objects by using the fine-grained hardware memory protection mechanisms (MPU), available in modern micro-controllers like the AURIX. It ensures the robustness and freedom from interference concept required by safety-critical applications.

Figure 1 sums up the workflow for generating parallel code for PXROS-HR:

- The user annotates an existing Scade 6 model to express potential parallelism (Section 2), using in particular sequential WCET information to decide how to partition the model.
- SCADE Suite KCG for Multi-Core generates C code as well as traceability information mapping the C code to the input model (Section 3).
- A target integration script generates code to allocate generated code to PXROS-HR tasks and/or cores and to implement communications (Section 4).

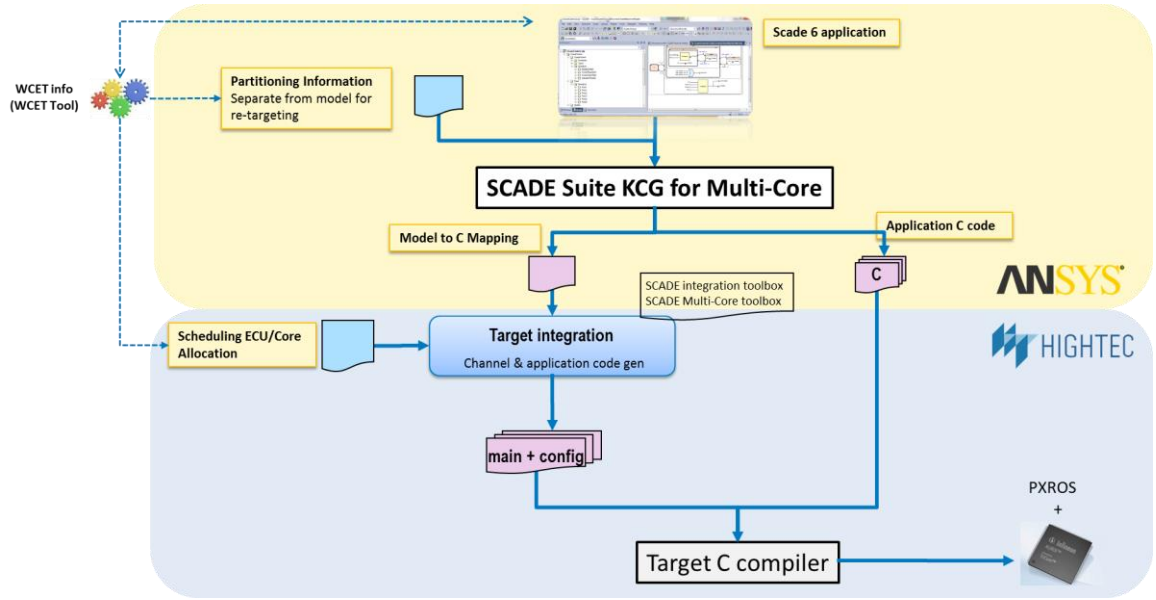

Figure 1 : SCADE Multi-Core Code Generation Flow

2. Parallelizing a Scade 6 model

2.1. An introduction to Scade

SCADE Suite is an integrated design environment for critical applications including model-based design, simulation, verification and qualifiable/certified code generation.

SCADE Suite has been used for more than twenty years to design critical software, such as flight control and engine control systems, automatic pilots, power management systems, rail interlocking systems and signaling, emergency braking systems, overspeed protection, nuclear power plant controls, ADAS in cars, and many other aerospace, railway, energy, automotive and industrial applications.

Applications are implemented using the formally defined Scade 6 language [Colaco & Al.] which is a synchronous data-flow language combining dataflow constructs as in Lustre [Caspi & Al.] with control-flow structures like

hierarchical state machines. Like other synchronous languages, Scade 6 divides time into discrete instants. A Scade model defines flows, that is sequences of values indexed by instants.

The SCADE Suite KCG code generator generates C or Ada code from a Scade 6 model and implements static analyses to ensure strong safety properties like determinism. The code generation and the static checks are qualified/certified for various safety standards (DO-178C/DO-330 at TQL-1, IEC 61508 at SIL 3, EN 50128 at SIL 3/4, and ISO 26262 software up to ASIL D). The code generated is an imperative transition function, called the *step function*, which computes from the previous state and the inputs (coming from sensors) a new state and the outputs (going to actuators). This function is called cyclically to react to changes of the physical environment.

2.2. Parallel Computation Model

The first step to execute an existing Scade 6 application on a multi-core Infineon AURIX platform is to identify parallelism in the model. As the language is naturally concurrent, any operations without data dependencies could be executed in parallel. Such fine-grained parallelism is however not realistic, so we require the user to specify how to partition the model for parallel execution.

Our approach proposes to parallelize the execution of the step function. This is done by grouping operator instances into *parallel subsets*. The instances in the same parallel subset can be executed in parallel in fork-join style. The instances in the same parallel subset must be independent, that is, the inputs of an instance cannot depend instantaneously on the outputs of another instance in the same parallel subset. This is checked statically by KCG and guarantees the absence of data-races or deadlocks at runtime.

Parallel subsets can be put anywhere in a Scade model. They can be put only in the small part of the model which is computationally intensive and can gain from parallel execution. Parallel subsets can be nested. We also allow to put instances located in different operators in the same parallel subset, as long as they end up in the same operator after expansion (expanding a Scade operator consists in replacing an instance with the body of the operator).

Our approach allows to parallelize an existing model without having to modify its architecture: the functional architecture of the model can remain independent of target integration matters. This is really the spirit of a model-based approach: describe *what* is computed, not *how* this computation is performed. The code generator takes care of splitting parts of the model into tasks ready for integration on the target.

Annotations can also be supplied from an external file, without any modification of the model. This can be useful to have different partitioning of the same model. In a qualification context, it can also avoid modifications of the model which would result in additional activities (eg. reviews). Such external file maps model paths of operator instances to their parallel subset.

Even with the facilities described previously, it may not be possible to parallelize an existing Scade model because of the sequentiality of the computations. A classic solution in that case is to use pipelining and to add memories between the different parts of the computation. It takes more steps to compute the result, but the different parts of the pipeline can be executed in parallel.

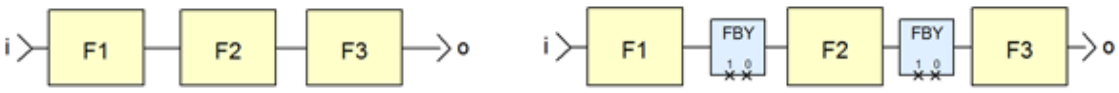

Figure 2: Pipelining with parallel subsets

Pipelining can be achieved the same way in Scade, by adding a unit delay (denoted `fby` in Scade), between different operators. This breaks the dependencies between the operators, so that they can be put in the same parallel subset. Figure 2 shows an example of such pipelining.

2.3. Illustration

We illustrate our approach using the ROSACE case study [Pagetti & Al.] which models a longitudinal flight controller. A Scade 6 version of this model is shown in Figure 3 (parallel subsets are displayed on the top-right corner of operator instances). The simplest way to parallelize the model is to find parts of it with no data-flow dependencies and with similar execution times. The sequential WCET tool provided by AbsInt and integrated into SCADE Suite can be used to identify CPU intensive parts of the model.

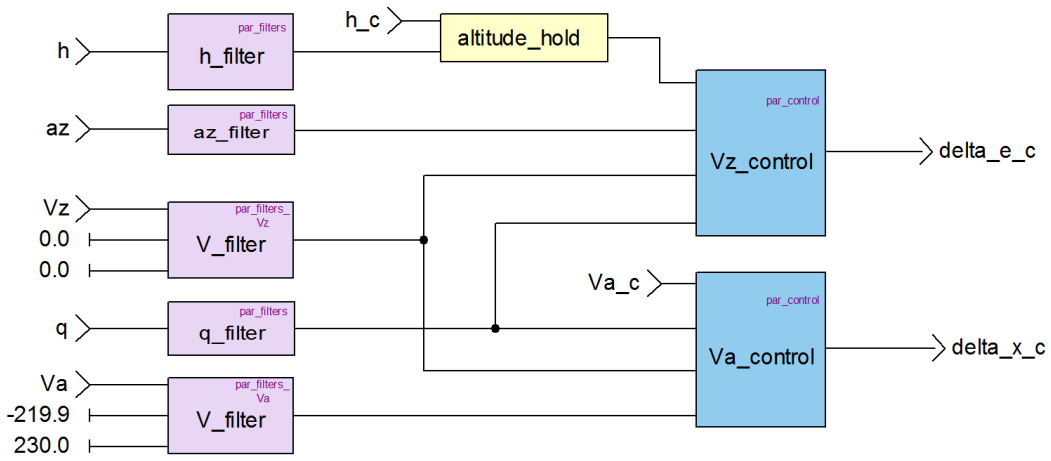

Figure 3 : ROSACE case study in Scade 6

The different filters (in purple) have no dependencies so they can be put in the same parallel subset named `filters`, and similarly for the two controllers (in blue) in another one named `control`. As instances from a parallel subset need to be independent, `h_filter` and `Vz_control` cannot be put in the same parallel subset.

3. Code generation

3.1. Generated code structure

The next step for the user is to generate C code using our KCG extension. It generates processes that communicate via one-to-one channels, that is, a Kahn process network (KPN) [Kahn]. One process executes the root operator of the Scade model. Several other processes, called workers, are generated: one for each instance in each parallel subset. A worker receives data on an input channel, call the operators, then sends the result on an output channel.

The behavior of the generated code is the same as for the sequential version, provided that the integration code correctly implements this KPN. The user does not need to test the behavior of the parallel version, only the implementation of channels and other usage conditions of KCG generated code. This is already done in the case of the PXROS-HR target.

The KPN model is common for multi-core execution of embedded systems and is also used by PXROS-HR. It guarantees isolation between processes, since there is no global memory and communications are done through channels. In the case of PXROS-HR, such a model is also independent of the allocation of processes to tasks and cores, since channel communication is abstracted through message sending and handled completely by the OS Kernel. The OS further provides physical data and task spatial separation needed in safety critical applications utilizing AURIX HW Memory Protection Unit.

3.2. C code API

The C code generated by KCG uses macros for all channel operations to remain target agnostic. The implementation of these primitives for a given platform must be provided by the integrator.

A channel is represented by a C structure with an additional field called `data` of a user-defined type `kcg_channel_data` which can be used for communication (eg. to store a semaphore or a lock).

```
typedef struct kcg_tag_az_filter_in_ch_type {
  kcg_float64 in1;
  kcg_channel_data data;
} az_filter_in_ch_type;
```

The following macros must be defined:

- `KCG_SEND(ch)`: sends on the channel the value stored in the structure fields
- `KCG_RECV(ch)`: receives the value from the channel and store it in the structure fields
- `KCG_DECL_SENDER(ch, t)`: declares a sender of channel `ch` of type `t`
- `KCG_DECL_RECEIVER(ch, t)`: declares a receiver of channel `ch` of type `t`

Section 4.2 shows an example of implementation of channels using shared memory and semaphores for synchronization.
There is actually one macro per operation per channel, so that each channel can be implemented differently. These macros are defined by default using the generic macros defined above.

Here is an example of the C code generated for the worker corresponding to `az_filter`:

```
KCG_DECL_RECEIVER_az_filter_in_ch(az_filter_in_ch_type, az_filter_in_ch)
KCG_DECL_SENDER_az_filter_out_ch(az_filter_out_ch_type, az_filter_out_ch)

void az_filter_worker(outC_az_filter_worker *outC)
{
  KCG_CHANNEL_RECV_az_filter_in_ch(az_filter_in_ch);
  /* _L9=(Controller::az_filter)/ */
  az_filter_Controller(az_filter_in_ch.in1, &az_filter_out_ch.y1, &outC->Context_az_filter);
  KCG_CHANNEL_SEND_az_filter_out_ch(az_filter_out_ch);
}
```

By default, each worker receives its inputs from the root process and sends its outputs to the same process. This can be quite inefficient if the root process only forwards this output to another process. In that case, KCG will optimize communications so that the second worker reads directly from the output channel of the first worker, without any synchronization with the root process. For instance, the worker executing `Vz_control` will directly read the output channel of the one executing `az_filter`.

4. Target integration

4.1. Integrating KCG generated code

The following steps should be performed to execute the code generated by KCG for multi-core:

- the workers must be allocated to OS tasks and cores. When developing critical software, it is important that the user keeps full control on the scheduling/mapping;
- the implementation of channel operations must be provided;
- the integration code must be written: it initializes the OS tasks, calls cyclically the root operator step function and handles communication with external software components.

Thanks to the Python *SCADE Multi-core Integration Toolbox* these steps can be fully automated.

4.2. Bare-Metal execution

The most straightforward integration is on bare-metal using threads and communicating with shared memory. In this configuration, channels are FIFOs of size one with one reader and one writer, which are one of the most basic data-structure of parallel programming. Here is an example of the implementation using Pthreads semaphores of size one:

```
#include <semaphore.h>
#include <errno.h>

typedef sem_t kcg_channel_data;

/* wait for the semaphore to be active (i.e. equal to 1) and then decrement it by one. */
#define KCG_CHANNEL_RECV(channel)                                \
  while (sem_wait(&(channel).data) != 0 && errno == EINTR) continue

/* make the semaphore active (i.e. equal to 1) */
#define KCG_CHANNEL_SEND(channel) sem_post(&(channel).data)

#define KCG_DECL_SENDER(t, n)   t n;
#define KCG_DECL_RECEIVER(t, n) extern t n;
```

We have also experimented more low-level approaches, where channel operations are implemented using C11 atomics [C11] or explicit cache operations on Kalray MPPA architecture [Dinechin & al.].

The Integration Toolbox and Multi-Core Toolbox can be used to generate the integration code. It takes as input an allocation of workers to threads, that is a list of workers to execute in each thread. Each thread calls the cycle function of its workers in turn and then repeats. In order to avoid deadlocks during the execution, the toolbox

checks that the order of threads is compatible with the data dependencies resulting from channel communications. The toolbox also detects if a channel reader and writer are allocated to the same thread. This is, for instance the case of the channel `az_filter_out_ch` in this example. In that case, there is no need to use synchronization primitives and the channel can be implemented as a global variable.

4.3. PXROS-HR integration

4.3.1. PXROS-HR Overview

PXROS-HR is a certified multicore hard-realtime operating system (RTOS). It is designed to meet ISO 26262 and IEC61508 requirements for safety-critical multicore embedded applications. It features a priority-driven preemptive task scheduler, flexible interrupt handler subsystem and event and message-based intertask/intercore communication.

PXROS-HR Tasks are self-contained independent embedded entities. Each task is executing a proprietary code over its proprietary protected resources, such as the task stack, data and communication objects. The protection is achieved by the OS Kernel, which is assisted by the HW Memory Protection Unit (MPU). The HW MPU guarantees that tasks can only access their explicitly granted resources, while an overall impact on the system performance is minimized. All the system components (task private resources, kernel and user objects) are thus each *individually* protected from illegal accesses, enabling:

- Freedom from interference (intrusion-free system) and Error containment (task faults are not propagated outside of the task context).
- Safe coexistence of generated certified model code, hardware-near-software components (drivers) and generic non-certified task code.
- Optimal system partitioning and effective collaborative code development and module verification.

The PXROS-HR multicore communication framework is based on a *non-shared memory* paradigm and *exclusive access* to data resources. It avoids the need of performance-intensive spinlock/mutex synchronization primitives or time-based intercore synchronization, to prevent data racing conditions. The PXROS-HR is thus a primary candidate to host a data flow driven applications running in a distributed multicore environment, such as the SCADE generated parallel code.

The Message is a primary means of data exchange among PXROS-HR tasks. Each message contains an arbitrary payload of data with exclusive *Access rights* (ownership). The access rights to the message data are passed from a sending to a receiving task during the atomic message transfer operation. At any moment, just one task in the system may acquire the exclusive access rights to the message to guarantee data integrity. The MPU assisted Kernel prevents any illegal access to the message content.

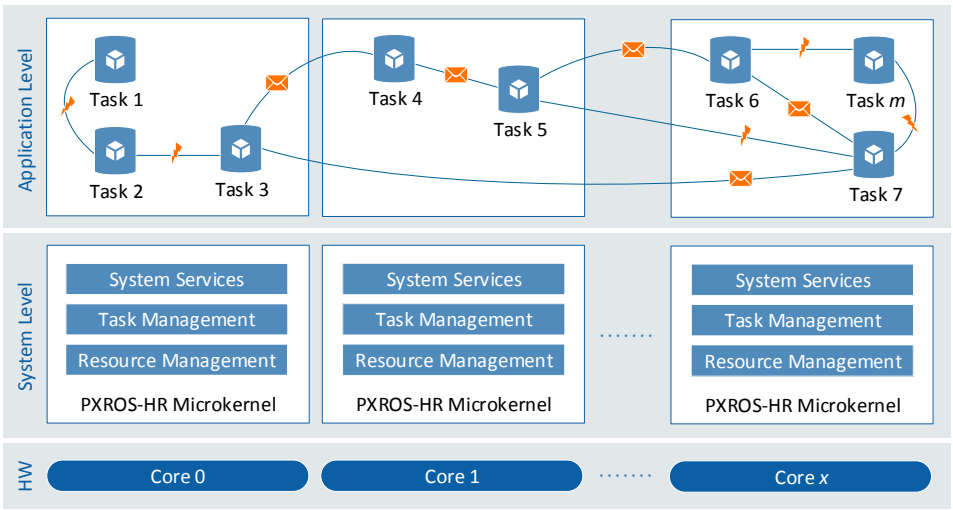

Figure 4: PXROS-HR multicore concept

The Message exchange is handled by the PXROS-HR Kernel and it is entirely core-agnostic. The same programming patterns apply to both single and multicore development. Tasks can be thus freely allocated to any available core. There is no need to modify the source code of a task when its execution needs to be moved from one core to another. This makes multicore development, load balancing, and verification a significantly easier process.

4.3.2. Integrating the KCG generated code

The Python target integration script developed by HighTec automates the generation of integration code for PXROS-HR, using the SCADE Integration Toolbox and Multi-Core Toolbox.

The integration script takes as input the allocation of workers to PXROS-HR tasks specified in the JSON file. By default, each worker is in a separated task, but several workers can also be placed in the same task. This can improve efficiency by removing useless communication if these workers communicate directly. The JSON file also provides other information vital for generation of the OS integration code, such as SCADE task priorities, number of available cores or stack size:

```
{
  "threads": [
    { "instances": [ "Controller::az_filter", "Controller::Vz_control" ], "core": 1,
      "tPrio": 25, "tStack": 256, "iStack": 16 },
    { "instances": [ "Controller::h_filter" ], "core": 1, "tPrio": 25, "tStack": 256,
      "iStack": 16 },
  …
  ],
  "number_cores": 3,
  "root_task": { "core": 0, "tPrio": 24, "tStack": 1536, "iStack": 32 }
}
```

The communication hierarchy and causal dependencies are resolved based on traceability information generated by KCG (mapping.xml file). Implementation of communication channel macros uses PXROS-HR native safe message send and message receive operations.

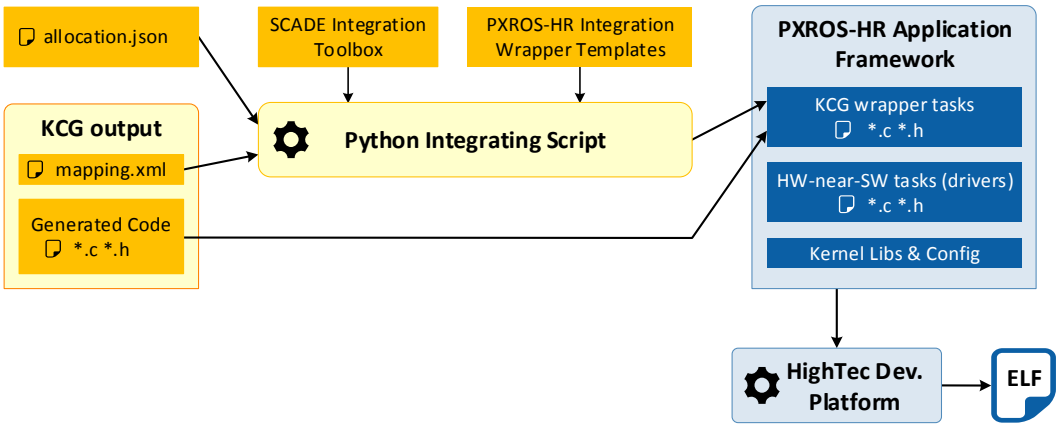

Figure 5: SCADE integration process with PXROS-HR

The output of the integration processing is a set of C source, header, and linker script files compliant to the PXROS-HR Application Framework task specification. Generated files can be directly built to produce a target executable image.

The following code snippets show actual PXROS-HR wrapping code as automatically generated by the Python Integration Script for the SCADE example illustrated in paragraph 2.3 and generally described in Section 3.

Communication Channels

The macro definitions for all the channels defined by the SCADE model are generated in a designated header file (*kcg_channel.h*):

```
#include "htc_pxutils.h"
…
#define KCG_CHANNEL_RECV(channel) {\
    extern PxMbx_t MBX_RECEIVER_##channel; \
    HtcChannelReceive_wait((PxMsgData_t)&(channel), sizeof((channel)), \
    MBX_RECEIVER_##channel);\
}

#define KCG_CHANNEL_SEND(channel) {\
    extern PxMbx_t MBX_SENDER_##channel; \
    extern PxMbx_t MSP_SENDER_##channel; \
    HtcChannelSend((PxMsgData_t)&(channel), sizeof((channel)), MBX_SENDER_##channel,\
```

```
    MSP_SENDER_##channel);\
}
…
#define KCG_CHANNEL_RECV_h_filter_in_ch(ch) KCG_CHANNEL_RECV(ch)
#define KCG_CHANNEL_SEND_h_filter_out_ch(ch) KCG_CHANNEL_SEND(ch)
…
#define KCG_DECL_RECEIVER_h_filter_in_ch(kcg_c1, kcg_c2)  static kcg_c1 kcg_c2
                                        __attribute__ ((section (".KCG_task2_bss")));

#define KCG_DECL_SENDER_h_filter_out_ch(kcg_c1, kcg_c2)   static kcg_c1 kcg_c2
                                        __attribute__ ((section (".KCG_task2_bss")));
…
```

The physical implementation of SCADE channels, specified by the KCG_CHANNEL_RECV() and KCG_CHANNEL_SEND() macros, utilizes the PXROS-HR kernel API to implement desired functionality based on native protected messaging mechanism. The following snippet illustrates the PXROS-HR code handling the channel communication for receiving and sending data:

```
void HtcChannelReceive_wait(PxMsgData_t chData, PxSize_t chSz, PxMbx_t mbx)
{
  …
  /* Sleep until a message is received */
  PxMsg_t msg = PxMsgReceive(mbx);
  …
  /* Open the message and pass its payload to the worker */
  PxMsgData_t msgData = PxMsgGetData(msg);
  PxBcopy((void*)msgData, (void*)chData, chSz);
  /* Release resource */
  msg = PxMsgRelease(msg);
  …
}

void HtcChannelSend(PxMsgData_t chData, PxSize_t chSz, PxMbx_t mbx, PxMbx_t msp)
{
  …
  /* Get pre-allocated message from the local mailbox */
  PxMsg_t msg = PxMsgReceive(msp);
  …
  /* Open the message and pass data from the worker as the message payload */
  PxMsgData_t msgData = PxMsgGetData(msg);
  PxBcopy((void*)chData, (void*)msgData, chSz);
  /* Send the message to the desired worker task */
  msg = PxMsgSend(msg, mbx);
  …
}
```

SCADE Tasks

For each entry in the *allocation.json* file, a PXROS-HR task is generated. The task which acts as an entry point to the SCADE topology is called the “Root”.

The structure of each task is similar:

- Initialization of all the PXROS-HR objects associated with communication channels of the SCADE workers that were assigned to the given task (that is, Mailbox objects for receiving channels and Message objects for sending channels).
- Call directly execution of the SCADE workers in the assigned order, typically in an infinite loop.

The code snippet for SCADE Task 2, encapsulating the “h_filter” worker follows:

```
…
#pragma section ".KCG_task2_bss" aw
PxMbx_t MBX_RECEIVER_h_filter_in_ch;
PxMbx_t MBX_SENDER_h_filter_out_ch;
PxMbx_t MSP_SENDER_h_filter_out_ch;
#pragma section
```

```
void KCG_task2(PxTask_t myID, PxMbx_t myMbx, PxEvents_t myActEv)
{
  …
  /*  Create mailbox for receivers and register names  */
  MBX_RECEIVER_h_filter_in_ch = HtcCreateMbxAndRegisterName(((PxNameId_t){{ … }}));

  /*  Query names and create message pool for senders  */
  MBX_SENDER_h_filter_out_ch = HtcQueryMbxName_wait(((PxNameId_t){{ … }}));
  MSP_SENDER_h_filter_out_ch = HtcCreateMsgPool(1,(PxSize_t)sizeof(h_filter_out_ch_type));

  /*  KCG task code  */
  outC_h_filter_worker outCtx_h_filter_worker;
  h_filter_worker_reset(&outCtx_h_filter_worker);
  while (1) h_filter_worker(&outCtx_h_filter_worker);
  …
}
```

Low-level task memory management

All the task data are MPU protected. Any attempt the task does in order to access the data outside its MPU context will trigger a system trap to maintain the freedom from interference principle.

The task stack, and thus all the task local data, are implicitly a part of its MPU context when the task is created. Task global and static objects, that are not part of the task stack, require explicit placement into the known memory location. This location becomes a part of the task MPU context too, and consequently accessible for the task.

To make this process transparent for the user, PXROS-HR enables to place global and static objects for each task in a memory section with a unique symbolic name. The section is specified using *#pragma* or *__attribute* directives. The previous code snippet of the SCADE Task 2 defines, for example, the section called *".KCG_task2_bss",* where several of the task global variables are located.

A formal task local linker script needs to be further provided, to link the task sections with the PXROS-HR build system. It enables the PXROS-HR framework to resolve the physical locations of these variables and include them automatically to the task MPU context when the task is created. The task local linker scripts are also automatically generated by the Python Integration Script together with the KCG task codes.

4.3.3. Deploying SCADE application in a mixed environment

On any embedded system, the autogenerated code will most likely need to coexist with other system components. The RTOS Kernel and its ecosystem, HW-related tasks (peripheral drivers) as well as other generic application tasks containing non-autogenerated code, such as communication stacks or human-machine interface, are among such components. This situation is illustrated in the following picture.

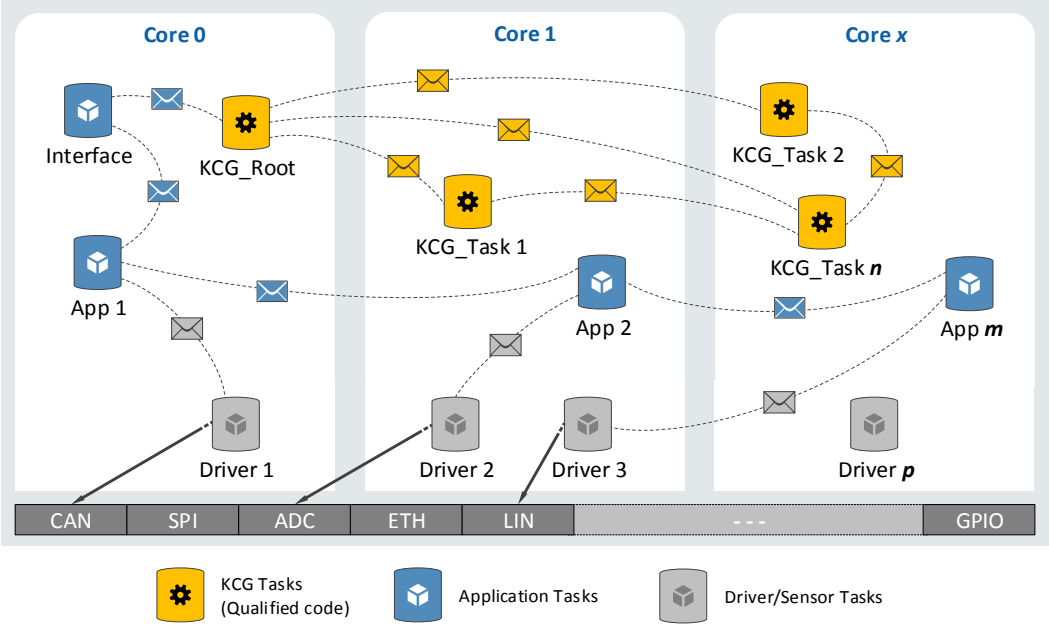

Figure 6: PXROS-HR multicore application structure

The PXROS-HR freedom from interference concept supports such intrusion-free coexistence. It effectively enables to separate tasks running the qualified code from tasks running a non-qualified code. It is also possible to deploy several SCADE models on the same embedded target when required, offering the same level of protection.

The inter-task communication is core-agnostic and task code does not require core-related dependencies. Tasks thus can be executed out of any core without code changes. This facilitates load balancing, development, and verification process as well as system scalability.

5. Conclusion

The flow presented in this paper allows generating multi-core code from any legacy Scade 6.6 application. Key properties of the current sequential code are kept in the multi-core generated code: execution determinism, static memory model and same behavior between the model, the sequential and the distributed code. Supporting a specific target is straightforward thanks to the execution model and the integration toolbox.

SCADE generated code and PXROS-HR concepts match perfectly. The combined flow brings a very high safety level from end to end and AURIX multicore performance is accessible to the non-expert.

The solution proposed here only considers the parallelization of the step function, so all threads synchronize at the end of each instant. A more ambitious issue is to allow the parallelization of tasks with different rates, eg. to allow a slow task to run in parallel with the rest of the program until its output is read several instants later. Combining our approach with the introduction of periodic clocks like in Prelude [Puffitsch & Al.] or n-synchrony [Mandel & al.] seems like a promising lead.

6. References

[Girault] A. Girault. A survey of automatic distribution method for synchronous programs. In *International workshop on synchronous languages, applications and programs*, *SLAP, volume 5, 2005*

[Lee & Al.] Lee, E. A., & Messerschmitt, D. G. (1987). Synchronous data flow. *Proceedings of the IEEE, 75(9), 1235-1245*

[Stuijk & Al.] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *DAC'07, pages 777–782. IEEE, 2007*

[Puffitsch & Al.] Puffitsch W, Noulard E, Pagetti C. Mapping a Multi-Rate Synchronous Language to a Many-Core Processor. In *RTAS'2013. 2013.*

[Cohen & Al.] A. Cohen, L. Gérard, and M. Pouzet. Programming parallelism with futures in Lustre. In *ACM International Conference on Embedded Software (EMSOFT'12), Tampere, Finland, October 7-12 2012*

[Colaco & Al] J.-L. Colaco, B. Pagano, and M. Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. *In TASE'2017.*

[Caspi & Al.] P. Caspi, Pilaud, D., Halbwachs, N., & Plaice, J. A.. LUSTRE: A declarative language for programming synchronous systems. *In POPL 1987.*

[Pagetti & Al] Pagetti, C., Saussié, D., Gratia, R., Noulard, E., & Siron, P. The ROSACE case study: From simulink specification to multi/many-core execution. *In RTAS'2014, 2014.*

[Kahn] G. Kahn, The semantics of a simple language for parallel programming, *Proceedings of IFIP Congress, 1974.*

[Mandel & al.] L. Mandel, F. Plateau, and M. Pouzet. Lucy-n: a n-Synchronous Extension of Lustre. *In MPC'10. 2010*

[C11] ISO/IEC 9899:2011. Programming language C, 2011.

[Dinechin & al.] de Dinechin, B. D., Ayrignac, R., Beaucamps, P. E., Couvert, P., Ganne, B., de Massas, P. G. & Strudel, T. A clustered manycore processor architecture for embedded and accelerated applications. In HPEC'13.