CERTAINTY

CErtification of Real Time Applications desIgNed for mixed criticaliTY

ICT COLLABORATIVE PROJECT

Multicore Code Generation for Time-critical Applications using BIP Tools

Due date of deliverable: M30 (31/04/2014)

Lead Beneficiary: 2 (UJF)

tool version 3.2: 23 September 2014

Table of content

1.1 Code Generation Tool Flow 3
1.2 Multi-core Code Generation from BIP 4
1.3 DOL Critical Frontend 5
1.3.1 DOL Language Support in BIP 5
1.3.2 DOL Language Extensions in BIP 6
1.3.3 Support for DOL Mapping and Scheduling 7
1.4 Tool Release 7
1.4.1 Website 7
1.4.2 Directory structure 7
1.4.3 Release history 8
1.5 Acknowledgements and Contact Information 8

1.1 Code Generation Tool Flow

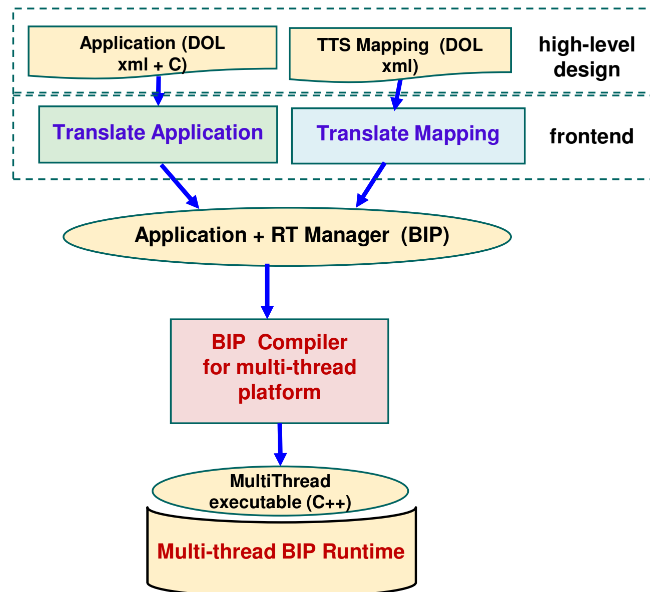

Figure 1 Code generation tool flow

In this document we give a brief overview of the publicly available code generation prototype tools developed in FP7 European project CERTAINTY, in Working Package 7.

The structure of the code generation flow is shown in the figure above. The top-part represents a high-level model-based design, serving as input to the tool. A model-based framework for *time-critical* (i.e. hard and firm real-time) applications should be used, in the figure we assume DOL Critical, a framework developed in project CERTAINTY. DOL provides the model of the application (task graph) and the model of deployment (mapping). The application model includes both the task communication structure (here: the DOL XML) and the task functions (here: the C files). The mapping file should contain an XML description of a statically partitioned schedule, i.e. a schedule where the tasks are statically bound ("mapped") to the cores.

The front end is a tool that translates the high-level models into equivalent BIP model. The BIP models define the executable semantics for all elements in the system, including the tasks and the schedulers. The BIP models can be both analyzed and executed, e.g. for simulation purposes.

The bottom part of the figure represents the code generation from BIP to multicore platform. Currently we implicitly assume a shared memory multicore platform where tasks can be implemented using POSIX threads or similar thread model. The BIP components are translated to C++ classes that can be orchestrated by multi-thread BIP runtime library. Note that the C++ language was used to facilitate the easy structuring of code together with data; we do not have a significant reliance on dynamic memory allocation, typical for many other C++ applications. In Section 1.2 we describe the multi-core code generation from BIP in more detail, whereas in Section 1.3 we describe on the DOL Critical frontend and Section 1.4 gives an overview of the tool release structure.

1.2 Multi-core Code Generation from BIP

The code is generated for multi-cores by translation from BIP. This is done by BIP compiler: `bipc` from the multi-thread distribution of BIP and requires the provided multi-thread library for BIP runtime. The multi-thread shared-memory model is assumed, implemented using POSIX multi-threading.

We have currently builds for two target platforms:

- standard Linux 64-bit (requires **gcc** at least v4.5 and Java at least v1.7)
- the MPPA™ platform of Kalray (www.kalray.eu) (with AccessCore™ at least 1.1)

The BIP runtime library for Linux is included in the tool release, but only for the purpose of prototyping, as we see no way to guarantee predictable real-time execution in this case. Still, it often works for simple examples with large real-time slack or small amount of parallelism. The library for the MPPA multicore platform is available on special request.

For the BIP language we use its *real-time* variant, which supports timed automata clocks (www.bip-components.com), with some important deviations:

- only rendezvous interactions are supported (no broadcast)
- the support of BIP priorities is limited
- timing constraints for internal transitions are not supported, they are scheduled as eager (i.e. urgent) transitions
- the semantics of internal transitions is different: they may take *any* time to execute ("*continuous*" transitions), as opposed to default instantaneous transitions (note, this is a new experimental feature).

By default, all atomic components run in a separate POSIX thread. However, in multicore architectures, with limited number of threads, the user is expected to calculate a static component-to-thread mapping at compile time and to provide it to `bipc` in the form of a "thread-map" text file via a command line option. Note that one separate thread is always reserved for the BIP runtime library. The thread file should be a two-column text file with (arbitrary) thread names in the first column and the component names in the second one. Compound components and hierarchical subcomponents are also supported (using "." in hierarchical paths).

A main guideline for the thread mapping is as follows. The "continuous" components (i.e., those having at least one "continuous" transition) should not be mixed with the "instantaneous" ones on the same thread. The continuous components would represent *computation tasks* and the instantaneous one would represent the *controllers* responsible for task activations.

The *controllers* should have negligible execution times, so they do not need parallelization and can be collected in one separate thread. The *computation tasks* can be mapped (partitioned) according to a partitioned multiprocessor scheduling algorithm, and the scheduling policy should be implemented a controller component called *runtime manager*. More information on this approach can be found in this research paper:

- D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, **Modeling Mixed-critical Systems in Real-time BIP**. In. *Proc. ReTiMiCs-2013*, *Workshop on Real-Time Mixed Criticality Systems*, pp. 29-34, workshop at RTCSA-2013.

Multi-threaded BIP runtime is presented in this paper:

- A. Triki, J. Combaz, S. Bensalem, and J. Sifakis. **Model-Based Implementation of Parallel Real-Time Systems**. In *Proc. FASE'13, Fundamental Approaches to Software Engineering*, LNCS, vol. 7793, pp. 235-249, Springer, 2013.

Examples: See "scheduled" examples in this tool release, where the "launch" script illustrates how multi-threaded applications equipped with a RT manager are executed in Linux on top of BIP runtime.

1.3 DOL Critical Frontend

The documentation of the DOL Critical language and tool suite is available from:

http://www.tik.ee.ethz.ch/~certainty/dolc.html

Same as the presented tools, it was first developed in CERTAINTY project.

Here we refer to DOL Critical simply as **DOL**, but it should not be confused with the previous "non-critical" version developed in earlier projects. We provide DOL2BIP frontend for multicore code generation. This frontend translates the DOL programs and the corresponding "TTS" scheduling policy into the BIP code that is ready as input for multi-thread code generation.

Refer again to Figure 1 in the first section. The DOL2BIP frontend is responsible for translating the DOL application and the "TTS" scheduling files into BIP components. Here TTS stands for *time-triggered with synchronization*, which is the scheduling method developed in CERTAINTY project in the context of DOL framework.

Finally, the generated BIP file is provided to the BIP compiler for multi-thread platform and linked with multi-thread BIP runtime.

Note that in DOL, the tasks are called *processes*, so we use terms "tasks" and "process" interchangeably in this document.

1.3.1 DOL Language Support in BIP

The DOL frontend in the BIP tools support a subset of the DOL language. Working on code generation, we gave priority to the DOL features that were judged to be most commonly used in the FMS application use case of CERTAINTY project. Our language limitations are mainly in the supported task activation patterns and DOL API functions.

1.3.1.1 Activation patterns

Supported:

- periodic pattern
- aperiodic pattern
- implicit pseudo-periodic - on top of aperiodic "protocol" feature

Not (yet) supported activation patterns:

- periodic with mode

- restartable
- explicit pseudo-periodic

1.3.1.2 DOL API functions

Supported API functions:

- DOLC_read
- DOLC_write

Not (yet) supported:

- DOLC_send_event
- DOC_yield

The latter two functions are required only for the activation patterns that are not yet supported.

1.3.2 DOL Language Extensions in BIP

BIP tools provide some relatively recent DOL extensions:

- aperiodic "protocol"
- precedence constraints between *any* two processes

The aperiodic protocol is a means to program the *activation times for aperiodic processes*. Two additional parameters are supported in the application XML:

- `protocol_period`
- `protocol_source`

The period is the minimal distance between the events in aperiodic burst. The protocol source points to a user-defined C source file that should contain a function that "decides" whether the process should be activated. This function is called periodically with the period specified as "protocol period".

In BIP tools for DOL, the precedence constraints can be enforced, using DOL syntax, between any two (a)periodic processes with any period, although the current prototype version of DOL may have only a limited support for that.

In addition to the default use scenario for precedence constraints, we believe that they can be also used at the phase of porting single-core real-time applications to multi-cores. The precedence constraints can reproduce the "rate-monotonic" or any other fixed-priority assignment settings, popular in hard-real-time software designs. Extra precedence constraints can be (temporarily) added during the functional testing phase, where both the multiprocessor application expressed in DOL and the reference fixed-priority implementation are run for comparison of the outputs. To this end, the user should put a precedence constraint between every two *communicating* processes, from the higher-priority process to the lower-priority process. If the processes do not communicate then the priorities do not matter for their functional behaviors. The extra precedences will enforce an execution order compatible with fixed-priority, enforcing sequential order over some pairs of tasks but not necessarily over the whole application.

Note that this may restrict the scheduling, so we recommend it only for functional testing phase.

1.3.3 Support for DOL Mapping and Scheduling

Next to the DOL Language, we provide support for the TTS mapping and scheduling provided in "DOL mapping" format in XML. The DOL2BIP frontend generates the BIP components corresponding to the basic elements of a TTS schedule. If requested via the command line by the user we create the thread-map text file, necessary for the BIP compiler to correctly partition the BIP components between the threads.

Note that here we completely follow our recommendations for thread mapping, formulated in Section 1.2. In particular, we put the *controller* components for task activation and inter-task communication channels in a separate *controller thread*, independent from the threads intended for the *continuous* components (i.e. the DOL processes).

From version v3.1 the tool supports TTS schedules with aperiodic processes. However, the TTS schedule support has a restriction that the *period* of the processes receiving the data from aperiodic ones be the same as the *interval* of the aperiodic process.

1.4 Tool Release

1.4.1 Website

Official website for this tool:

http://www-verimag.imag.fr/Multicore-Time-Critical-Code,470.html

1.4.2 Directory structure

The top part of the directory tree of the tool release is presented commented below.

dolc2bip/RT-BIP

`bipc` compiler and 64-bit Linux runtime of single-threaded RT-(real-time) BIP

this BIP version is used for **functional simulation**

dolc2bip/MULTI-BIP

`bipc` compiler and 64-bit Linux runtime of the multi-threaded RT-BIP

this BIP version used for **real-time execution**

dolc2bip/tool_v3.2

Contains dol2bip frontend tool itself and the examples

ATTENTION! Check the README file there to carefully *configure* the tool scripts.

dolc2bip/tool_v3.2/examples

Contains two groups of examples: functional (only DOL language) and scheduled (TTS scheduling included)

Each example has a script "**run**" for single-thread functional simulation

Some scheduled examples have "**launch**" script for real-time multi-thread execution.

1.4.3 Release history

v3.0 – April 2014 – first public release

v3.1 – August 2014

bug fixes in the dol2bip tool (some internal steps were not « eager », port data was written in the "up" part of a connector, which is forbidden, etc.)

added tool support for TTS schedules with aperiodic processes, introduced support for multiple invocations of a process in the same TTS frame.

fixed uninitialized memory bug causing the multi-thread engine to spontaneously start in lazy mode leading to huge « clock drift » immediately at start (reported by THALES on Kalray MPPA architecture)

v3.2 – September 2014

Fix of segmentation fault in multithreaded engine encountered in FMS use case.

Aperiodic process: Fix of multiple clocks in the "sporadic" generator (to support pipelined generation); Fix of the case when aperiodic has smaller precedence than the user process connected to it.

1.5 Acknowledgements and Contact Information

Supervisors and Contact Persons:

Prof. Saddek Bensalem, UJF-Verimag

Saddek.Bensalem@imag.fr

Dr. Marius Bozga, CNRS-Verimag:

Marius.Bozga@imag.fr

UJF-Verimag Development team:

Jacques Combaz, Ahlem Triki: multi-threaded RT-BIP runtime and compiler

Petro Poplavko: BIP extensions for CERTAINTY

Paraskevas Bourgos: DOL-Critical Frontend

Dario Socci: BIP prototypes and components for mixed-critical systems

Acknowledgements

We would like to thank Anakreontas Mentis, from UJF-Verimag, for valuable help in porting the BIP runtime on the MPPA platform.

We would also like to acknowledge valuable contribution of Georgia Giannopoulou, from ETHZ, in tool testing and discussions.