ICRA 2010 Workshop on

Innovative Robot Control Architectures for Demanding (Research) Applications

– How to Modify and Enhance Commercial Controllers –
Abstract

— Control architectures may not be the hottest topic in robotics research but without a solid and powerful control architecture the hottest topics cannot be tackled. —

Numerous demanding robotics applications and research projects employing commercial manipulators require high-rate low-level interfaces to the robot hardware. Often the performance of robotic systems and the quality of research results can be improved significantly by employing an appropriate low-level control interface to the robot hardware instead of the original controller. The controllers provided by manufacturers often lack high-rate low-latency communication interfaces – if a low-level control interface and sufficient computing power are provided at all. Modifying commercial robot controllers or even developing open robot controllers for commercial manipulators requires expert knowledge in several fields. Unfortunately, concrete implementations are hardly described in the literature and thus this endeavor poses a high obstacle for many research groups who are not primarily concerned with electronics and control engineering.

These workshop proceedings provide hands-on advice on how to implement open control interfaces for commercial manipulators of several popular manufacturers – respecting legal restrictions. Moreover, they address both hardware and software issues such as interface design, suitable control approaches, and the lower layers of control architectures, which significantly influence the performance.
List of Topics

• Open high-rate low-level control architectures (ABB, Comau, KUKA, Motoman, Sensable, Stäubli)
• Low-level control interfaces: paradigms and implementation issues
• Manipulator control software architectures
• Distributed real-time middleware solutions
• Open robot control software
• Multi sensor integration
• Software development and code generation

Organizers

Daniel Kubus  
TU Braunschweig, Germany  
Institut für Robotik und Prozessinformatik  
d.kubus@tu-bs.de

Klas Nilsson  
Lund University, Sweden  
Dept. of Computer Science  
klas@cs.lth.se

Rolf Johansson  
Lund University, Sweden  
Dept. of Automatic Control  
Rolf.Johansson@control.lth.se
Contributions

- **Open Software Architecture for Advanced Control of Robotic Manipulators in Non-Structured Environments** [1]
  
  *Authors:* Javier Gámez García, Juan Gómez Ortega, Alejandro Sánchez García, and Silvia Satorres Martínez

  *Keywords:* Open software architectures; robotic manipulator control; robot sensing systems; multisensory systems

  *Affiliation:* University of Jaén, Jaén, Spain

- **The Fast Research Interface for the KUKA Lightweight Robot** [2]
  
  *Authors:* Günter Schreiber(1), Andreas Stemmer(2), and Rainer Bischoff(1)

  *Keywords:* Low level control interface for industrial controllers; position, impedance, and torque control; real-time control: 1 ms

  *Affiliation:* (1)Kuka Roboter GmbH; (2) DLR, Institute of Robotics and Mechatronics, Germany

- **Commercial Controllers Enhancements and Open Source Robot Control Software: Addressed Solutions for Demanding Applications** [3]
  
  *Authors:* N. Pedrocchi, M. Malosio, F. Vicentini, L. Molinari Tosatti, and G. Legnani

  *Keywords:* Open-source realtime operating system; modular software architecture; force based/model based control algorithms; human-robot interaction

  *Affiliation:* Institute of Industrial Technologies and Automation, National Council of Researches, Italy

  
  *Authors:* Gianluca Antonelli(1), Stefano Chiaverini(1), Valerio Perna(2), and Fabrizio Romanelli(2)

  *Keywords:* Open and modular architecture; optimized trajectory planning; sensor fusion; high-level programming

  *Affiliation:* (1) Dipartimento di Automazione, Università degli Studi di Cassino, Italy; (2) Research & Development Group, Motion and Control, Comau Robotics SpA

- **Low-Level Control of Robot Manipulators: Distributed Open Real-Time Control Architectures for Stäubli RX and TX Manipulators** [5]
  
  *Authors:* Daniel Kubus, Alexander Sommerkorn, Torsten Kröger, Jochen Maaß, and Friedrich M. Wahl

  *Keywords:* Hardware and software architectures/modifications; synchronization approaches in distributed control systems; characteristics of the Stäubli low-level interface; real-time middleware

  *Affiliation:* TU Braunschweig, Institut für Robotik und Prozessinformatik, Germany

- **Low-Level Control of Robot Manipulators: A Brief Survey on Sensor-Guided Control and On-Line Trajectory Generation** [6]
  
  *Authors:* Torsten Kröger(1) and Friedrich M. Wahl(2)

  *Keywords:* Distributed real-time computing, on-line trajectory generation, force/torque sensor integration, camera integration, hybrid switched-system control, manipulation primitives

  *Affiliation:* (1) Artificial Intelligence Lab, Dept. of Computer Science, Stanford University, USA; (2) TU Braunschweig, Institut für Robotik und Prozessinformatik, Germany

- **Open Architecture xPC Target Based Robot Controllers for Industrial and Research Manipulators** [7]
  
  *Authors:* Matjaž Mihelj and Marko Munih

  *Keywords:* Open architecture robot control; robot low level control interfaces; software tools and code generation; robot control and haptic library, robot manufacturers: Stäubli, Motoman, Moog FCS (HapticMaster), Sensable (Phantom), Matlab xPC Target

  *Affiliation:* Faculty of Electrical Engineering, University of Ljubljana, Slovenia
• Flexible Application Development and High-Performance Motion Control Based on External Sensing and Reconfiguration of ABB Industrial Robot Controllers [8]
  Authors: Anders Blomdell(1), Isolde Dressler(1), Klas Nilsson(2), Anders Robertsson(1)
  Keywords: Robot sensing, control, actuation, interfaces, applications
  Affiliation: (1) Dept. of Automatic Control, (2) Dept. of Computer Science, Lund University, Sweden

• Motion Planning and Monitoring of an Industrial Robot via a Virtual Manufacturing Cell [9]
  Authors: Fabrizio Abrate(1), Basilio Bona(1), Marina Indri(1), Daniele Messa(1), and Aldo Bottero (2)
  Keywords: Virtual robotic cell, bidirectional HW/SW architecture, motion planning, robot monitoring
  Affiliation: (1) DAUIN - Politecnico di Torino, Italy; (2) Comau Robotics SpA

• Model-Driven Robot-Software Design Using Template-Based Target Descriptions [10]
  Authors: Jan F. Broenink, Marcel A. Groothuis, Peter M. Visser, and Maarten M. Bezemer
  Keywords: Robot Software Architectures; Model-Driven Design; Code Generation; Software Tools
  Affiliation: Control Engineering, Faculty EE-Math-CS, University of Twente, The Netherlands
## Contents

1. **Open Software Architecture for Advanced Control of Robotic Manipulators in Non-structured Environments** by J. Gámez García et al.  
2. **The Fast Research Interface for the KUKA Lightweight Robot** by G. Schreiber et al.  
3. **Commercial Controllers enhancements and Open Source Robot Control Software: addressed solutions for demanding applications** by N. Pedrocchi et al.  
5. **Low-Level Control of Robot Manipulators: Distributed Open Real-Time Control Architectures for Stäubli RX and TX Manipulators** by Kubus et al.  
6. **Low-Level Control of RobotaManipulators: A Brief Survey on Sensor-Guided Control and On-Line Trajectory Generation** by T. Kröger et al.  
7. **Open Architecture xPC Target Based Robot Controllers for Industrial and Research Manipulators** by M. Mihelj et al.  
8. **Flexible application development and high-performance motion control based on external sensing and reconfiguration of ABB Industrial Robot Controllers** by A. Robertsson et al.  
9. **Motion planning and monitoring of an industrial robot via a virtual manufacturing cell** by M. Indri et al.  
10. **Model-driven robot-software design using template-based target descriptions** by J. Broenink et al.
Model-Driven Robot-Software Design Using Template-Based Target Descriptions

Jan F. Broenink*, Marcel A. Groothuis*, Peter M. Visser**, Maarten M. Bezemer*

Abstract—This paper is about using templates and passing model-specific information between tools via parameterized tokens in the generated, high-level code, to get a better separation of design steps. This allows for better quality of the models and more reuse, thus enhancing the efficiency of model-driven design for the (industrial) end user. This is illustrated by the realization of the embedded software of a real system.

We conclude that reuse is easier. However, the presented method can be generalized more, as to connect to other tools and platforms.

I. INTRODUCTION

Designing embedded control software is often done using models from which code is generated (i.e. model-driven). Besides models of the embedded control software, also models of the dynamic behavior of the robot mechanism to be controlled, are used for design and verification purposes.

We use a layered architecture for the design of the embedded control software, whereby parts are specified as reusable, configurable building blocks (See Fig. 1). This supports separation of design activities in the different design steps, thus stimulating focus per design step, and allowing different design steps be conducted simultaneously. It furthermore enhances efficiency, as generic parts need to be coded only once (as code templates), and used many times.

As a result of the usage of multiple models and tools for the design the embedded control software, the code generation and integration phase becomes a bit complicated without proper measures. The desired goal is a full code generation path, which does not require manual adaptation/additions to the generated code in order to make it suitable to run on a specific robot target. Therefore the used tools are chained to support step-wise and partial code generation.

Other design flows, like Matlab / Simulink / Real-Time Workshop xPC target [1] or Labview Robotics [2], are restricted to vendor-specific tools and targets, and do not follow the strict separation between development steps, resulting in models polluted with target-hardware-specific information.

This contribution explains the usage of code templates together with token replacement techniques and a supporting toolchain as means to generate the implementation code from the design models and to run this code on a variety of own designed and industrial hardware platforms.

The toolchain is designed in such a way that it allows a flexible interconnection of various tools (both commercial and academic) used for different design steps, while ensuring the separation of concerns between these steps.

Section II presents some background knowledge on the formalisms, techniques, and tools used. In Section III, we present the method (details are in [3]), and in Section IV, we briefly illustrate it with an example. Section V combines discussion and conclusions.

II. BACKGROUND

A typical mechatronic system like a robot consists of a combination of a mechanical (physical) system, mixed-signal and power electronics, and an embedded (motion) control system (ECS). The combination of a mechanical setup and its ECS software requires a multi-disciplinary and synergistic approach for its design, because the dynamic behavior of the mechanics influences the behavior of the software and vice versa (also known as cyber physical systems). Therefore we adhere a mechatronic or systems approach for the design of the physical system and its software, because they should be designed together (co-design approach) to find an optimal and dependable realization.

A. Formalisms

Formalisms are the languages and syntax used for embedded control systems modelling. Two main core Models of Computation (MoC) are used to describe the total mechatronic system behavior (plant, control, software and I/O): a continuous-time MoC (dynamic system, control) and a discrete-event MoC (software). The modelling language used for the continuous-time MoC is the bond-graph notation, which is a domain-independent notation for describing dynamic systems behavior [4], [5]. The used modelling formalism for the embedded software is a layered dataflow driven architecture modeled in the Communicating Sequential Processes (CSP) algebra [6], [7]. The CSP algebra allows us to provide a mathematical proof of the correctness of our software designs with respect to deadlocks, livelocks and refinements.

B. Tools

The commercial modeling and simulation tool 20-sim [8] is used to model dynamic systems and for controller design (the continuous time part). 20-sim supports multi-disciplinary modeling with library components for many engineering disciplines and also supports the domain independent bond-graph notation. It also supports model checking.
and has an extensive control toolbox and has a customizable
template-based C-code synthesis facility for automatic code
synthesis of whole models or submodels (e.g. only the con-
troller) with a strict separation between model dependent
and target dependent code. For the design of the embedded soft-
ware architecture in which the controllers will be integrated
(discrete-event part), we have developed a graphical CSP
drawing tool, gCSP [9]. This tool is based on the CSP process
algebra and allows us to explicitly define both communication
(rendezvous data-flow), composition (sequential, parallel,
concurrent execution), timing and priorities of the designed
process network in a single model. It has animation facilities
for verification purposes and code generation facilities for
both CSP formal checking, using the CSP model checker
FDR2 [10], and C/C++ for the final implementation on
a CPU or an FPGA (programmable hardware with real
parallelism).

C. Techniques

To support separation of concerns between design steps
and to support a

To accomodate partial code generation and a smooth
chaining of tools, each modeling tool uses a template-based
code generation mechanism, with tokens to mark the places
where the model-specific information must be added. The
used approach is similar to the pipe and filter pattern [11,
Ch. 2].

These code-templates are building blocks containing to-
kens that represent model-specific information. The model-
to-code transformation uses token replacement to generate a
piece of code (e.g. only a control algorithm). We call this
partial code generation, because the generated code is in
general not finished, i.e. not all tokens are replaced in one
run: The token replacement is used as step-wise refinement
technique, where tool A generates code for tool B that is
not necessarily complete. In the incomplete sections, tokens
are placed that can be refined by tool B. Tool B may insert
additional tokens and only replace some of the tokens in the
code from tool A. Tool C can be used for the next refinement
and this process needs to be continued until all tokens are
replaced.

Typically a token in code has a distinct separate syntax de-
notation to reflect that is not yet complete code. For instance,
in C-code, a token can be denoted as %TOKEN_NAME%. This
notation is invalid C-code and the C-code will not compile
without errors if the token is not replaced, serving as a check
on completeness of the token replacement activities.

III. METHOD

A. General Approach

We use a model-driven design method for designing the
embedded control software (ECS), with a close cooperation
between the involved disciplines. For the ECS, we use a
layered architecture, inspired by [12] with layers for: user-
interfacing, supervisory control, sequence control (order of
actions), loop control (control law and motion profiles),
safety purposes and measurement and actuation. See Fig.

1. The ECS is a combination of an event-driven part and
a time-triggered part with different and often challenging
(real-)time requirements for the different layers. Hard real-
time behavior is for example required for the last two layers.
The control laws for the loop control layer require a periodic
time schedule with hard deadlines in which jitter and latency
are undesirable.

Concurrent design techniques are used to shorten the total
time from idea to realization. In general, the continuous-time
part (loop control, M&:A, physical system, see Fig. 1) and
the discrete event part (supervisory, sequence control, UI)
are designed concurrently and in different tools, as the MoC
are quite different [13]. Of course, the interfaces between
these parts need jointly be specified. The design process
of each part consists of a stepwise refinement process and
a alternatives checking process (design space exploration)
which are applied alternately until the model has sufficient
detail, such that for the software parts, all code can be
generated.

While designing the loop controllers, the starting point
is a physical system model (a model of dynamic behavior
of the robot mechanism). From this model, the control
engineer derives the required control algorithm, based on
the assumption of an ideal target, in for instance 20-sim [8].
The next step is to incorporate target behavior (discrete time,
AD/DA effects, signal delay, scaling) via stepwise refinement
into the design before the loop controllers can be integrated
in the ECS design.

B. Embedded Control Software Implementation

The next step after the control law design step, is a further
refinement of the controllers towards code. The essence is
to strictly separate these design steps. By doing so, in the
loop controller design models, no artifacts dealing with the
specific computer target (i.e. interfacing connections) ap-
pear. However, generic computer implementation issues, like
discrete-time computation for the control laws, continuous-
time to discrete-time connection points and rough estimations
of I/O delays, are needed, as these influence the control
algorithm.

This strict separation implies that the models in each
step only consist information needed for the specific step
only. The advantage of this is that (1) reuse of models / model parts becomes easier; (2) testing different targets to
implement one set of controllers is straightforward to do;
(3) updates of targets can be tested easily; (4) using virtual
prototypes and test equipment (i.e. rapid prototyping with

![Fig. 1. Embedded control system and its software architecture](image-url)
mostly more powerful target computers) can now be done quite systematically.

The Way of Working of the ECS implementation and testing is as follows:

1) High-level code generation: The control algorithm is translated into C-code. (also called “Synthesis Export”).
2) Connect: The inputs and outputs of the controller need to be connected to the inputs and outputs of the target.
3) Compile: The code must be (cross-)compiled into an executable application. This is sometimes called “backend compilation” (C to embedded), to distinguish it from the earlier compilation phase.
4) Deploy: send the “application” to the target.
5) Run: start the application (Also stopping and pausing may be required.)
6) Monitor: For validation purposes, it should be possible to compare the on-target results with the simulation. Hence monitoring and data logging capabilities are needed.
7) Modify: Altering (controller) parameters on-the-fly: during test runs it should be possible to allow for the last fine-tuning in combination with the plant.
8) Import run-time data: Signal log data and execution traces can be imported in the modeling tools for validation purposes and further analysis.

This Way of Working is one cycle in the iteration of rapid prototyping, and the Run - Monitor - Modify is a iteration
cycle on itself.

The Connect activity is the crucial step to ensure the strict separation between the controller design and ECS implementation phases.

This strict separation is supported by our tool chain. The connection of input / output variables of the controller specification to the output / input signals of the target hardware is done in a separate tool (a target connector), i.e. outside the modeling, simulation and controller design tool (see Fig. 2). This requires that the ECS generated by the design tool contains tokens recognized by the target connector. These tokens are specified in the code templates of the modeling tool, as indicated in Fig. 2. Furthermore, the target connector needs the interface specification of the target (i.e. which input and output signals are available). In the target connector we use (20-sim 4C, [8]), in which the interface specification is available via the target template (i.e. a target configuration file). The C-code generated from the modeling tool (20-sim) has tokens, which are interpreted by the target connector.

After connecting the model variables to the hardware I/O signals, the code can be generated by the target connector (i.e. the tokens produced by 20-sim are substituted by the connection information), cross-compiled and loaded onto the target. Note that this is the second code generation activity.

In our case, the target connector relies on a small daemon process running on the target, which transfers commands between the ECS running on the target and a command interface running on a (development) workstation, which is part of the target connector tool. Besides starting / stopping the ECS, it has facilities to monitor and log signals on the target. As such, it is a rapid prototyping tool, and provides steps 2 to 7 of the Way of Working.

Note, that the approach we use here is in principle tool-independent. However, the code generation facility of a tool must provide means to generate the tokens. These tokens are specific keywords or keyword – value pairs, delimited with a specific character (currently a %). The target connector interprets these keywords and fills in the choices the user has made when connecting the model variables to the target signals.

IV. EXAMPLE: CARTESIAN PLOTTER DESIGN

This section describes the stepwise realization of the ECS for a real system, to show how we use the desired method, techniques and tooling in practice. The example is a Cartesian plotter (see Figure 3). More information about this plotter can be found in [13] The steps from models to the final ECS target realization are shown in Fig. 4.

Fig. 5 show the top-level 20-sim model that is used for the controller design and dynamics simulations. It contains simple setpoints (e.g. square and circle drawing), the control algorithms, basic knowledge about the I/O and a model of the plotter dynamics. Fig. 6 shows the internals of one of the major components of the plotter, namely the Y-axis. The “controller” submodel has the required functionality for the “loop control” layer in Fig. 1.

Because 20-sim is a (dynamics) modeling and simulation tool and not a software design tool, the other ECS layers are designed separately in gCSP (see Fig. 7 for the top level model). The 20-sim controller submodel is translated into C-code following step 1) of the Way of Working using a code
Mechatronic co-design test case 25. Work on cartesian plotters and co-simulation. Analysis results are presented in section III. The main differences when mechanics, software, and micro-mechanical components [11] are involved in a co-design are mixing discrete events and control engineering interfaces to C and VHDL not being standard. A lack of portability is noted, due to the need of time synchronization. Bouchet et al. [10] connect geographically distributed simulators and languages. The area of unlinked tools are used. When the system validation. The main differences when mechanics worked. Branicky et al. [16] provides a framework that makes Giano more promising as an open framework.

In mechatronic system design often a collection of unlinked tools are used. When the system needs to be considered at the same time [18]. This is within the boundaries of a landscape A4 page.

In traditional design approaches the different parts of a development phase. Another tool, Microsoft worked. For this purpose, a co-simulation between the ECS software and the plant model is needed and that the designed safety layers are functioning. This can be tested via a co-simulation between the ECS software and the plant (simulated plotter) using the connections as shown in Fig. 8.

Similarly, processor-in-the-loop simulation and hardware-in-the-loop simulation (step 3b in Fig. 4), can elegantly be supported: the controller model always stays the same.

Replacing the targeted processor board by another type (e.g. an ARM board) or the usage of a different I/O board requires only a different target template and no changes in the design running RTAI real-time Linux, with which experiments can be performed (step 4 in Fig. 4). Preferably, these experiments were tested first in simulation, such that the results of the controller controlling the real setup can be compared with the simulation (step 8 of the Way of Working).

The advantage of the strict separation between the two models and the target code details, becomes clear when we follow the stepwise refinement route in Fig. 4. Before testing the generated ECS software on-target, we want to make sure that the combination of the two software pieces is working fine and that the designed safety layers are functioning. This can be tested via a co-simulation between the ECS software (gCSP) and the plant model in Fig. 5 (step 3a in Fig. 4). For this purpose, a co-simulation target template can be used in the Target connector to redirect the I/O from the software to the 20-sim model with the plant (simulated plotter) using the connections as shown in Fig. 8.

Subsequently following the Way of Working results in code running on the target (a PC/104 embedded computer, generation template with 20-sim model tokens and a gCSP process interface. The gCSP model in Fig. 7 is also translated into C-code with tokens and a netlist. These two pieces of code are generated separately and have still no detailed I/O information or target information.

The next step is to read the code and netlists in the Target Connector, select the wanted target and connect the dangling inputs and outputs to the target I/O (e.g. driver calls). The Target Connector processes the code and replaces I/O and target tokens by the corresponding code pieces from its target template (step 2 in Fig. 2).

Subsequently following the Way of Working results in code running on the target (a PC/104 embedded computer,
models, unless details like delays and resolution change. In that case the I/O details in the 20-sim model needs a small adaptation because these changes can influence the controller performance. This can be resolved by using a library with I/O building blocks (board support package submodels) which simplifies the model changes into replacing 1 submodel by an alternate implementation.

V. DISCUSSION AND CONCLUSION

We expect that the approach presented here can also be used for commercial platforms and commercial robots, provided that the information needed in the target template can be obtained (i.e. distilled from the robot documentation). Furthermore, different tools can be used, provided that the model-specific information via the tokens can be passed between the tools used.

We conclude that the principle of exploiting templates and passing model-specific information via token replacement contributes to the separation of concerns / goals between the control algorithm design step and ECS implementation step. This separation of concerns supports quality enhancement of the models, and, as a result of that, supports better reuse of existing model (parts), thus raising design efficiency (i.e. reducing design time). However, we did not yet quantify the advantages of our approach over classical methods.

Ongoing work is done on supporting our approach on commercial industrial control platforms like the Bachmann M1 Controller hardware [14].

Future work is to generalize this approach by connecting other tools and execution platforms (e.g. OROCOS), and check whether also templating the target execution daemon contributes to better models and more reuse, thus being effective for the users (i.e. industry).

REFERENCES