A Software Architecture for Model-Based Programming of Robot Systems

Michael Geisinger, Simon Barner, Martin Wojtczyk, and Alois Knoll

Abstract While robot systems become more and more elaborate, the need to simplify programming them grows as well. Regarding the high degree of internal heterogeneity in the sense that different microcontroller platforms, protocols and performance layers are used in a single robot application, it is not feasible to have specialists dedicated to each individual task. This motivates the need for tool support that allows an abstract view on a robot's sensors and actuators, means to program individual components as well as to define their interaction. In this work, we present how the model-based development and code generation tool EasyLab can be extended to support programming of all parts of a robot, including the main controller as well as peripheral devices like smart sensors. We show three typical use cases in the context of mobile platforms and highlight EasyLab's advantages in each domain.

1 Introduction

Complex robot systems often consist of highly heterogeneous components and communication protocols. A typical example is the TAMS Service Robot (TASER) presented in [17], whose parts are connected via Ethernet, CAN, RS-232 and others. Developing software for each component individually is a time-consuming process. A better approach is to specify the system components at a high level of abstraction. This specification is condensed in form of a *model*, which in turn can consist of various parts: the *device model* specifies the hardware components, their interfaces and dependencies. The *application model* formalizes the actual task to execute. Finally the *distribution model* specifies how software components are deployed in a distributed environment and how they interact with each other.

Michael Geisinger · Simon Barner · Martin Wojtczyk · Alois Knoll
Technische Universität München, Boltzmannstr. 3, D-85748 Garching bei München, Germany
e-mail: {geisinge, barner, wojtczyk, knoll}@in.tum.de

The model-based code generation tool EasyLab [3] uses this modular strategy to allow specification of application requirements and behavior at a high level of abstraction. It currently supports device and application models. The integrated code generation engine can be used to transform the models into executable machine code optimized for the respective target platform.

EasyLab was originally designed for software development of *mechatronic systems* and includes components for mathematical computations, control systems (e.g., PID controllers) and communication over media typically used in industrial environments. However, due to its extensibility, EasyLab is also suitable for application in the field of robotics. This work will show how EasyLab can be extended to allow programming of all parts of a robot system. We show that different *modes of model execution* are a crucial point in this context and present uniform software interfaces for application deployment, debugging and monitoring of such systems. On the one hand, EasyLab can be used to generate code for the firmware of microcontroller systems. On the other hand, it also provides engine-driven execution of programs on the robot's controller system (based on an operating system or some kind of middleware).

In this paper, the term *programming* denotes the overall process of designing and implementing the control software of robot systems, whereas *deploying* means transferring the application to the *target* system. Since the latter one is our point of view, we also refer to it as the *local* system (in contrast to *remote* systems).

The rest of the paper is structured as follows: First, we summarize some related work in the domain of robot software architectures. In section 3, we show the main features of the model-based development tool EasyLab and point out the uniformity of some of our software architecture's interfaces in section 4. After three illustrative examples in section 5 we conclude with a summary and a section about future work.

2 Related Work

The usage of EasyLab in the domain of robotics requires to compare it to existing frameworks that tackle the challenging task of programming robot systems. We distinguish between different levels of abstraction:

- At the lowest level, a multitude of libraries have been created for robot systems to perform tasks like mathematical computations for kinematics, dynamics and machine vision [6, 10, 12]. EasyLab already provides this functionality partially through components for the mobile robot platform Robotino® introduced in section 5.1. In contrast to the approaches mentioned above, EasyLab also allows to specify required hardware resources of each component, which makes it possible to avoid inconsistent access to shared hardware already at the modeling level.
- In the field of robotics, middlewares are often used to hide complexity regarding inter-component communication. OpenRTM-aist [1] is a CORBA-based middleware for robot platforms that uses so-called robot technology components (RTCs) to model distribution of functionality. The Orca project [5] is a

component-based framework for building robotic systems, which provides abstraction layers for communication and focuses on reuse of software. The Player component of the Player/Stage project [9] provides a network interface to a variety of robot and sensor hardware based on TCP/IP. While EasyLab itself is not a middleware, it provides concepts to integrate them. On the one hand, this involves design and implementation of components, which is currently researched by integrating OpenRTM in such a way that the application logic of RTCs is modeled in EasyLab. On the other hand, support for distribution models is needed, which is subject to future work.

- Skill and behavioral based robot programming deals with the question how tasks and goals can be described intuitively and at a high level of abstraction. URBI [2] is a universal robotics software platform with built-in support for parallel execution and event-based programming. CLARAty [15] is a framework for generic and reusable robotic components and consists of a functional layer and a decision layer, whereas the latter allows high-level specification of tasks to execute. Until now, EasyLab was primarily applied to the field of automation and therefore only data-flow and state-based task specifications and adequate execution concepts were required. However, EasyLab's component-based architecture and its language primitives for explicit parallelism form a possible approach for the integration of the high-level concepts mention above.

The Ptolemy project [8] has a broader scope and is not restricted to the field of robotics. It provides a wide range of models and modes of execution for the design of real-time systems, which is also one of the goals of EasyLab. In addition to that, EasyLab explicitly models the target hardware and provides means to generate hardware-dependent code for different platforms from the same model.

3 EasyLab

EasyLab is a model-based development tool for the software component of embedded systems. It is part of the EasyKit[1] project, which aims at providing a methodology for the efficient development of *mechatronic systems* (i.e., systems with a tight interaction of mechanics, electronics and software) using hardware/software co-design. While *EasyKit* provides a modular hardware construction kit for the assembly of hardware prototypes of mechatronic systems, *EasyLab* allows to define software models for interaction with the respective hardware and features both a template-based code generator (the current output language is C) and an engine-driven model execution mode. Models can be specified in EasyLab using visual programming languages (see section 3.1). Using predefined primitive components from an integrated library, a user does not have to write a single line of code in order to develop an application for a mechatronic system. EasyLab can also be used

[1] See http://easykit.informatik.tu-muenchen.de/.

This work is funded by the German Ministry of Education and Research under grant 02PG1241.

to monitor the running system and tune certain parameters over serial lines or bus communication protocols such as Modbus [14].

3.1 Application Model

One of EasyLab's primary goals is to facilitate the way of programming target systems. For this purpose, it currently supports two visual modeling languages (for details see [3]):

1. **Structured flow chart**: The SFC language describes the states of a program, how state transitions are performed and which actions are taken when the program is in a certain state (typically execution of a data flow program as specified below). The SFC language was designed in the style of EN 61131-3 [11] (part "SFC").
2. **Synchronous data flow:** The SDF language describes a directed multigraph where each node is the instance of a certain actor type (also called *function block*). An actor type is defined by a set of typed input and output connectors as well as internal state variables. Furthermore, the actions **start()**, **step()** and **stop()** define an actor's effect on the input data and calculation of the output data. Edges in the SDF graph denote the data flow between actor instances. The SDF language was designed in the style of EN 61131-3 [11] (part "FBD") and also supports multi-rate data flow.

3.2 Device Model

In EasyLab, not only the application logic is modeled, but also hardware-related aspects of the system (e.g., the microcontroller being used). This is especially important for code generation, as low-level driver code must be generated differently for different types of hardware platforms. EasyLab's device descriptions are modular and can be nested to obtain composed devices (e.g., a robot system) from primitive devices (such as specific controllers, sensors and actuators). The library of primitive devices can also easily be extended.

Device behavior is defined by specifying handlers for the following actions: **open()** and **close()** are called when a connection to a device instance is to be established or dropped, respectively. **start()** and **stop()** are called just before a device will be used or when it should go "idle". **read()** and **write()** are called when data should be requested from or forwarded to the hardware (these operations are performed all at once for all data to ensure consistent sensor values and actuator control signals).

3.3 Modes of Execution

Once the application and device models are specified within EasyLab, they can be executed in different ways. The first approach is to generate code (currently: C code) from the models that is then executed *natively* on the target platform. If an operating system is available on the target, the model can also be transferred to the target platform and directly interpreted (*local execution* or *model interpretation*). A third possibility is to execute the model directly within EasyLab and exchange control signals with the target (*remote execution* or *model simulation*), which requires an adequate proxy application running on the target.

Figure 1 shows EasyLab's modes of model execution in correlation with different application domains. Some execution modes imply the existence of an operating system (OS). The graph also shows rough boundaries for the clock speed of the respective processor.

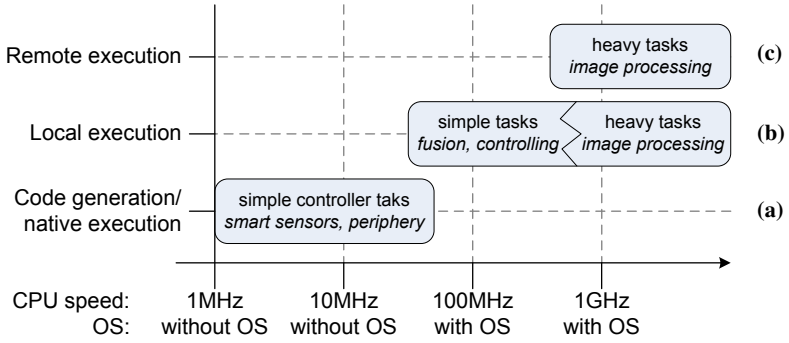

Fig. 1 EasyLab modes of execution against requirements and typical usage scenarios in italics. **(a)** Code generation and native execution fits best the needs of simple controller applications as found in smart sensors. **(b)** Local execution can run on any target with operating system. **(c)** Using remote execution, the task is simulated on a host PC and control signals are exchanged with the hardware.

3.3.1 Code Generation and Native Execution

The process of code generation and the provided interfaces are shown in figure 2. It is currently used for small applications without operating system (typically 8-bit microcontrollers). In the field of robotics, such systems are used to implement *smart sensors or actuators* that perform some kind of additional processing. EasyLab can be used to model the application logic of these systems and also the communication backend that allows other components to access them (see section 5.3 for an example). The template-based code generation approach provides the flexibility to generate code for different platforms such as middlewares or operating systems (see [3] for details).

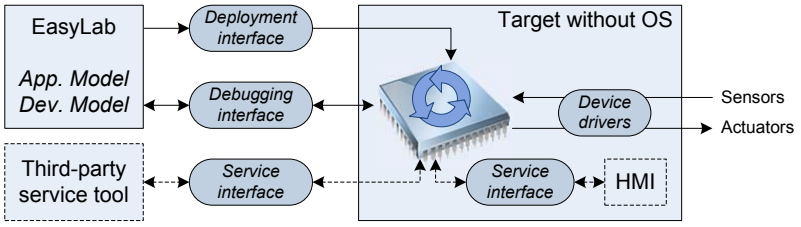

Fig. 2 Native execution use case: The compiled image is sent to the target via the deployment interface. The target executes the program (indicated by circular arrows) and interacts with the hardware using device drivers. A debugging interface enables inspection and step-wise execution during development. The optional service interface can be used to maintain and tune the system after deployment or to visualize and modify data using a human-machine interface (HMI).

EasyLab currently supports (but is not restricted to) the following compilers: `avr-gcc` for Atmel AVR[2], `mcc18` for PIC 18F[3] and `fcc907s` for Fujitsu F16[4]. Generating code for SFC programs is implemented using the continuation-passing style pattern. For SDF programs, static scheduling [4, 13, 16] is used to ensure efficient execution and predictable timing. Schedules for parallel periodic tasks (e.g. multiple SDF programs running at the same time) can be computed in advance using the concept of logical execution time [7]. However, programs cannot be modified after deployment without a new code generation run, which is different in the execution concepts described in the following sections.

3.3.2 Local Execution

By the term "local" we refer to execution of a program on the target hardware (i.e., on the robot). In contrast to code generation, local execution is performed by "interpreting" a program modeled in EasyLab using a console-mode version of EasyLab (i.e., a specially crafted version without graphical user interface). Figure 3 illustrates this use case.

An operating system is used to reduce the complexity of the interpreter and facilitates the implementation of tasks like host-to-target communication, model storage and dynamic loading of shared libraries. Interpretation of SFC programs is straightforward. For SDF programs, a dynamically linked library is specified for each actor type that provides the following C++ interface: The **start()** method is used to initialize the actor. The **step()** method is called each time the respective actor is executed according to the schedule. A **stop()** method is used to finalize the actor. The libraries for each actor are cross-compiled for the respective target's operating system and dynamically loaded on demand using a plug-in system. The same approach is also used

[2] http://www.nongnu.org/avr-libc/

[3] http://search.microchip.com/searchapp/searchhome.aspx?q=SW006011

[4] http://www.fujitsu.com/us/services/edevices/microelectronics/microcontrollers/datalib/softune/

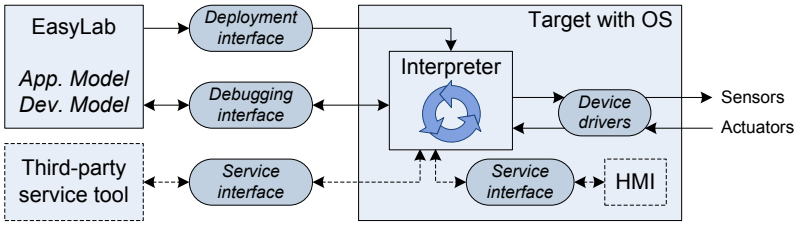

Fig. 3 Local execution use case: the complete application model is transferred to a console-mode version of EasyLab (interpreter) running on the target system. Debugging and service interface serve the same functionality and use the same communication protocols as in native execution mode.

for device drivers. This execution concept also allows to change the program while it is running by updating or retransmitting the model and the necessary plug-ins.

3.3.3 Remote Execution

Remote execution means that the application is not executed on the actual target system, but instead on a host PC running EasyLab (usually the machine where the application is being modeled). The target runs the console-mode version of EasyLab and only acts as a proxy that forwards sensor readings to and awaits control signals from the host. Figure 4 illustrates this configuration.

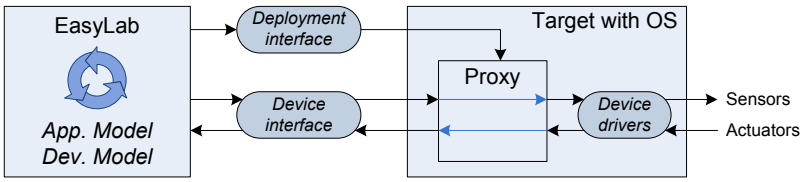

Fig. 4 Remote execution use case: The console-mode version of EasyLab running on the target is used to translate between requests from the host PC (on the left) and the actual device drivers. Deployment is realized by transferring device descriptions and drivers to the target. A service interface is not available on the target.

An advantage of this setup is that changes in the application model are immediately reflected in the behavior of the system. This makes it easy to modify or extend the program while it is already running. Only if the underlying device model is changed, the device proxy running on the target needs to be exchanged (redeployment). Another advantage is that the performance of the application is better if the processor of the host PC is more powerful than the one in the target system.

Note that remote execution could also be applied to target platforms without operating system. This would require to implement an adequate device interface and has not yet been researched.

4 Interfaces

As pointed out in [3], expandability is one of the primary design goals for EasyLab that is achieved by a clear separation of models, code-generation templates, execution and also GUI plug-ins. In addition to that, the concept of interfaces abstracts introspection and manipulation of execution instances. Since these interfaces are also used on embedded targets (native execution), the current implementation is based on C (a C++ version that integrates with the existing plug-in architecture is planned).

Access to variables is performed via unique identifiers. Possible implementations are the variable's RAM address (if known in advance) or arbitray identifiers that are mapped to the variable using look-up tables. The latter implementation is required in remote execution mode and for native execution in conjunction with dynamic memory allocation (i.e., when a variable's address can not be known in advance).

4.1 Debugging Interface

The debugging interface is only available during development. It is used to visualize and manipulate the current state of execution and all variable values directly in the graphical representation of the model and consists of the following functions:

getProgramId() retrieves a unique number that is used to match the model with the executed program instance. **getFlags()** retrieves variable flags (e.g., read/write access). **read()** retrieves the current value of a variable, while **write()** sets its value. There is support for transactions (atomic execution of multiple read and write operations). A transaction is initiated with **beginTransaction()** and executed on **commit()**. **getCurrentStack()** retrieves the execution stack and is used to determine and visualize currently active model items. **command()** sends commands to the target to run, break, step or reset it and to set or remove breakpoints.

4.2 Service Interface

In contrast to the debugging interface, this interface allows to monitor a running application and to change selected parameters of deployed programs (i.e., in "release mode"), which is a requirement for the use of programs generated by EasyLab in the field of automation. The *local service interface* can be used to connect to the running application from the target side. For native targets (code generation) this is implemented by linking another application into the final executable such as a human-machine interface (HMI). The *remote service interface* can be connected to over any communication medium supported by the target. An application scenario in context of the EasyKit project is to have a third-party service tool running on a mobile device to perform maintenance, diagnosis and benchmarking tasks.

5 Applications

In this section, we give three examples where model-based development using EasyLab is applied in the context of robot systems. They demonstrates how the different modes of execution pointed out in section 3.3 can be applied while using the same abstract types of device and application models at the topmost layer.

5.1 Mobile Robot Platform Robotino®

Robotino® is a mobile robot platform mainly used in education and research. It is not only equipped with a drive system allowing for omni-directional moves as well as distance, infrared and inductive sensors, but also features a camera with VGA resolution. Figure 5 (a) shows the default configuration.

The robot is equipped with a PC104 controller with a clock speed of 500 MHz (there are however newer versions with faster CPUs). The robot runs a real-time Linux operating system and can be controlled remotely over wireless LAN. The console-mode version of EasyLab runs directly on the robot (*local execution*, see section 3.3.2). However, the limited computational capacity of Robotino® prevents the implementation of extensive tasks like complex image processing, which is why the robot is mostly operated in *remote execution* mode (see section 3.3.3). For this purpose, the device model components in EasyLab establish the connection to the target. A library of customized SDF function blocks is available for reading the robot's sensor values and controlling its movements. To make this possible, a proxy for interactions with sensors and actuators is running on the robot. Typical applications are camera-based navigation (e.g., line following), induction-driven fine positioning and distance-based collision avoidance.

An advantage of this approach is its flexibility: users can easily explore the robot's functionality and intuitively understand how changes to the program are reflected in the robot's behavior. As the basic hardware components of the robot are fixed, the proxy application does not need to be adapted when modifying the program. The software Robotino®View is a special version of EasyLab that only supports remote execution with a fixed hardware model.

5.2 F5 Platform

The F5 platform is another mobile platform for which EasyLab device models have been defined. It is shown in figure 5 (b). In comparison to Robotino®, this platform is not only intended for education and research, but also for carrying out service tasks. It has larger proportions and is prepared to be extended by a robot arm to increase its versatility and ability to manipulate the environment. In previous projects, we demonstrated a robust and reproducible sample management process in a biotech

pilot plant [18] carried out by an autonomous, mobile manipulator which is going to be succeeded by the F5. Precise movements and gentle device interactions are assured by the utilization of a camera for image processing and a force/torque sensor attached to the tool of the manipulator to prevent damages.

Since the platform is equipped with a rather powerful computer (standard PC running at 2 GHz and a touch screen), it is suitable to perform both *remote* and *local execution* of arbitrary EasyLab models such as image processing, localization, mapping and path planning. A typical scenario for *local execution* is the previously mentioned sample management process, which is carried out autonomously by the mobile platform. Alternatively, a remote user may connect to the platform and utilize the *remote execution* of EasyLab models to carry out surveillance tasks and lab walkthroughs.

5.3 Smart Sensors and Actuators

Modularization is a long-standing concept in system design that fosters reuse of well-tested components and thus both improves quality and helps to cut cost. Hence, also in the area of robotics decentralization and the need for local ”intelligence” for both sensors actuators is becoming more and more important. The term *smart sensors* refers to ordinary sensors that have been augmented with local preprocessing (like calibration, statistics, ...) and offer the additional benefit of a simple interface to clients. The term *smart actuator* is intended to transfer this principle to the domain of actuators (e.g., a drive systems with integrated control unit). For many of these components, already a microcontroller offers enough computational power. In the course of an integrated development process, code generation and *native execution* as described in section 3.3.1 can be used to implement the necessary signal processing and control software. Hence, the same level of abstraction can be used for all computation units of a robot.

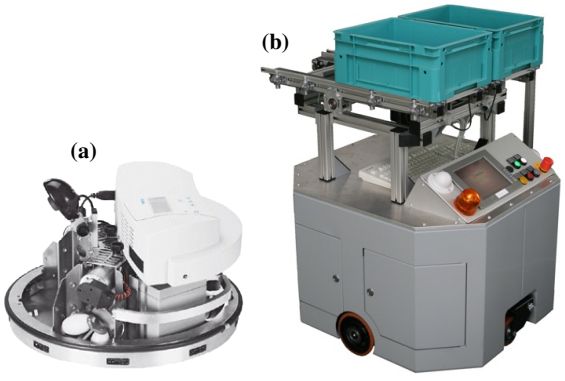

Fig. 5 Applications; **(a)** mobile robot platform Robotino®; **(b)** F5 platform in the current stage of expansion. Note that the images are not at the same scale. Images courtesy of Festo Didactic and REC GmbH.

As an example for a smart sensor application, we set up a PT100-based temperature sensor that was enhanced with a microcontroller and external EEPROM. The sensor provides calibrated and preprocessed data over Modbus. An example for a smart actuator is the "intelligent" pneumatic cylinder detailed in [3].

In both setups, we used the debugging interface to verify the application model during development. Compared to conventional smart sensor and actuator code development, the high level of abstraction fostered readability of the programs, facilitated debugging and thus reduced the potential for errors.

6 Summary

In this paper we argued that model-based development is a possible solution to the heterogeneous controller and communications architectures that are widespread in the field of robotics. While the raised level of abstraction hides implementation details and thus enables a more efficient and less error-prone development process, the approach followed in EasyLab still yields implementations that are adopted to the respective target platform. This is achieved by a threefold notion of model execution, namely native, local and remote execution, which were illustrated by three demonstrator implementations. The concept that is actually used depends on factors like processing power of the target system, presence of an operating system and whether tuning or altering the program while it is running should be supported.

7 Future Work

EasyLab's support for programming robot platforms is still to be extended in various directions. One of the most important features is *distributed modeling*, which will allow multiple components of a robot or networked system to be modeled within one EasyLab model. Data exchange can then be expressed at a very high level of abstraction by specifying producers and consumers of data. We consider to use the time-triggered approach because of its well integration with the SDF language. Going one step further, tasks such as acquiring and processing sensor data as well as controlling actuators can be distributed in the style of a *service-oriented architecture* (SOA), where each component offers services to the other components. An extension that goes in a different direction is support for *multicore architectures*. As both the SFC as well as the SDF language provide the possibility to explicitly model parallel execution of tasks, it is meaningful to distribute the load of these tasks in an intelligent manner among different computation cores, if available.

References

1. Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., Yoon, W.K.: RT-middleware: Distributed component middleware for RT (robot technology). In: International Conference on Intelligent Robots and Systems 2005 (IROS 2005), pp. 3933–3938 (2005)
2. Baillie, J.C.: URBI: towards a universal robotic low-level programming language. In: 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005), pp. 820–825 (2005). DOI 10.1109/IROS.2005.1545467
3. Barner, S., Geisinger, M., Buckl, C., Knoll, A.: EasyLab: Model-based development of software for mechatronic systems. In: IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications. Beijing, China (2008)
4. Bhattacharyya, S.S., Buck, J.T., Ha, S., Lee, E.A.: Generating compact code from dataflow specifications of multirate signal processing algorithms. IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications **42**, 138–150 (1995)
5. Brooks, A., Kaupp, T., Makarenko, A., Oreback, A., Williams, S.: Towards component-based robotics. In: Proc. of 2005 IEEE/RSJ Int. Conf. on Intellegent Robots and Systems (IROS'05), pp. 163–168. Alberta, Canada (2005)
6. Bruyninckx, H.: Open robot control software: The OROCOS project. In: Proceedings of 2001 IEEE International Conference on Robotics and Automation (ICRA'01), vol. 3, pp. 2523–2528. Seoul, Korea (2001)
7. Buckl, C., Regensburger, M., Knoll, A., Schrott, G.: Generic fault-tolerance mechanisms using the concept of logical execution time. In: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing), pp. 3–10. IEEE (2007)
8. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. In: Proceedings of the IEEE, vol. 91 (2003)
9. Gerkey, B.P., Vaughan, R.T., Howard, A.: Most valuable player: a robot device server for distributed control. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1226–1231. Wailea, Hawaii (2001)
10. Gourdeau, R.: A robotics object oriented package in C++. Ph.D. thesis, École Polytechnique de Montréal (2001)
11. International Electrotechnical Commission: Norm EN 61131 (2003)
12. Kapoor, C.: A reusable operational software architecture for advanced robotics. Ph.D. thesis, The University of Texas at Austin (1996)
13. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–35 (1987)
14. Modbus IDA: Modbus application protocol specification v1.1a (2004). URL http://www.modbus.org/
15. Nesnas, I., Wright, A., Bajracharya, M., Simmons, R., Estlin, T.: CLARAty and challenges of developing interoperable robotic software. In: Proceedings of 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003), vol. 3, pp. 2428–2435 (2003). DOI 10.1109/IROS.2003.1249234
16. Oh, H., Dutt, N., Ha, S.: Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In: Asia and South Pacific Conference on Design Automation, p. 6pp. (2006)
17. Westhoff, D., Zhang, J.: A unified robotic software architecture for service robotics and networks of smart sensors. In: Autonome Mobile Systeme 2007, pp. 126–132. Springer Berlin Heidelberg, Kaiserslautern, Germany (2007)
18. Wojtczyk, M., Marszalek, M., Heidemann, R., Joeris, K., Zhang, C., Burnett, M., Monica, T., Knoll, A.: Automation of the complete sample management in a biotech laboratory. In: S. Abramsky, E. Gelenbe, V. Sassone (eds.) Proceedings of Visions of Computer Science, BCS International Academic Conference, pp. 87–97. The British Computer Society, Imperial College, London, UK (2008)