Hiding Real-Time: A new Approach for the Software Development of Industrial Robots

Alwin Hoffmann, Andreas Angerer, Frank Ortmeier, Michael Vistein and Wolfgang Reif
Institute of Software and Systems Engineering
University of Augsburg, Germany
{*alwin.hoffmann, angerer, ortmeier, vistein, reif*}@informatik.uni-augsburg.de

***Abstract*— The application of industrial robots is strongly limited by the use of old-style robot programming languages. Due to these languages, the development of robotic software is a complex and expensive task requiring technical expertise and time. Hence, the use of industrial robots is often not a question of technical feasibility but of economic efficiency. This paper introduces a new architectural approach making available modern concepts of software engineering for industrial robots. The core idea is to hide the real-time critical robot control from application developers. Instead, common functionality is provided by a generic and extensible application programming interface and can be easily used. Hence, this approach can lead to an industrialization of software development for industrial robotics.**

I. Introduction

Compared to traditional production machines, industrial robots are designed to be flexible and adaptive. Given an appropriate tool, robots can be programmed to perform a large variety of tasks ranging from industrial manufacturing to medical applications [1]. For example, robots are able to perform quality assurance with optical sensors, assist humans in manufacturing or adapt their behaviour according to external sensing and environmental constraints. However, most applications where industrial robots are in use today share two characteristics: (1) mass production and (2) simple tasks. The most prominent example is the automotive production. In this domain, industrial robots are integrated into large-scale production systems to perform repetitive tasks like assembly, welding, or painting. In contrast, the use of industrial robots in aeronautics or in small and medium enterprises, where applications require robots to perform specialized and fast changing tasks, is rare.

The main reason for this limitation is the high programming effort which is necessary for adapting industrial robots to such tasks. Today, industrial robots are still programmed with specifically developed robot programming languages. These languages are derived from early imperative languages like ALGOL or Pascal and have not evolved much since then. Due to these low-level languages, programming an industrial robot is a complex task requiring considerable technical expertise and time. Because even trained users are hardly able to change and adapt robot programs, industrial robots are usually equipped and programmed to perform only one specific task. For example, an effort of six person months for programming a robot to weld a new car series, which is produced one million times, may be acceptable. But the same effort is not acceptable for a company that manufactures fast changing products in small production batches. It is worth mentioning that this problem becomes more important the more complex the tasks are. Hence, the use of industrial robots is often not a question of technical feasibility but of economic efficiency.

On the other hand, software development in general has changed much during the last 15 years. It is evolving more and more into an engineering discipline with the systematic use of methods and tools to develop, operate and maintain software [2]. The intention is to reduce the complexity of software development and to improve efficiency and quality. The application of software process models structures the development of software covering activities like requirement analysis, software design, implementation and maintenance. In object-oriented development, real-life concepts are modelled as closely as possible using software objects. Modern programming languages like C# and Java include strong type checking, reflection, exception handling and automatic memory management. A broad variety of libraries is available for those languages. Furthermore, integrated development environments have been created to assist programming teams in the creation of complex applications.

This paper outlines a concept for an integrated software architecture for programming and controlling industrial robots. The main goal is to make available the methods and tools of modern software engineering for the development of software for industrial robots. Techniques like high-level programming (e.g. by demonstration), more natural human-machine interfaces (e.g. using speech or gesture recognition), and easy-to-use applications with intuitive graphical user interfaces [3] often need to be adapted to a particular domain or even a specific problem. Considering the current software architectures with their proprietary, old-style programming languages, it is clear that the development of appropriate software is complex and expensive. Hence, the transition towards a modern software architecture has to be seen as an enabling technology. Efficient, fast and affordable software development is a key requirement for a wider distribution of industrial robots especially in small and medium enterprises.

The structure of the paper is as follows. Section II describes the core idea and motivates the proposed architecture. Subsequently, in Section III, the software architecture is

introduced in detail. An example is given in Section IV illustrating the benefits of this approach. In Section V, related approaches are presented. Finally, conclusions are given in Section VI.

II. Core Idea

Industrial robots are able to perform tasks with an impressive performance regarding speed, repeatability and accuracy. One reason is of course the excellent mechanical design of industrial-strength manipulators. However, the control design and its implementation is as important. A key requirement for the implementation of control methods where time constraints are important is programming of real-time systems. If certain computations exceed a defined barrier, the resulting behaviour is indeterministic and can lead to a system failure with unacceptable consequences. For example, a manipulator can follow a desired trajectory only by synchronized motions of all its joints. The exact duration of each single movement must be identical every time a program is run. Deviations from a given trajectory can result in severe damage. Therefore, hard real-time constraints and deterministic behaviour are major requirements for industrial robotics. Real-time is also important for controlling tools during movements (e.g. a glueing gun has to be turned on at a precise position) or the communication between cooperating robots (e.g. load-sharing).

However, applications for industrial robots are not necessarily real-time critical over the whole duration of their execution. The analysis of a broad variety of typical industrial applications (e.g. glueing, welding, palletizing) has shown that these applications embody only a small set of real-time critical tasks. A task can be considered real-time critical, if a coordinated and synchronized execution of multiple control actions is necessary. A single control action is always real-time critical and needs to be deterministic and precisely repeatable. For example, glueing a sequence of straight lines without stopping is a real-time critical task. The control actions of this task are the operation of the glueing gun and the linear movement of the robot. If control actions do not need to be synchronized, they can be considered as single independent tasks (e.g. moving the glueing gun from one seam to another along certain auxiliary positions). The main application only controls the proper execution of several tasks and is usually not real-time critical (e.g. the work flow of a glueing application).

This observation has two consequences. Firstly, applications for industrial robots can be mainly developed using standard technologies and environments. For example, modern general-purpose programming languages like C# and integrated development environments can be used. Secondly, it is even possible to hide robot control and real-time programming from application developers. Common functionality of industrial robotics can be provided by a well-defined and extensible application programming interface, and can be easily used by application developers. In consequence, they are able to focus on solving domain-specific problems and, as far as possible, do not have to deal with robot control problems and real-time constraints.

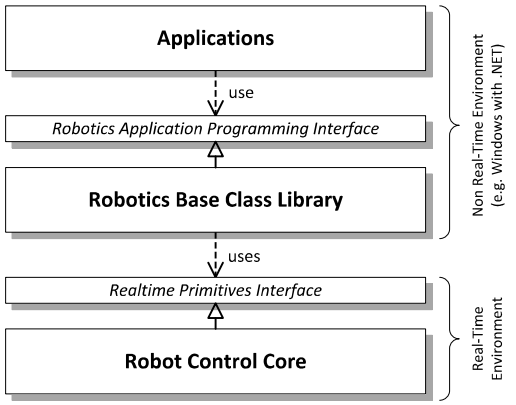

Fig. 1. The components of the proposed three-tier software architecture

From this point of view, software development for industrial robotics is changing in the same way as software development for business applications was changing a decade ago. Today, standard business applications are developed on top of modern frameworks like Microsoft .NET or Java Enterprise Edition. These frameworks incorporate solutions that software engineering has developed for many standard problems like synchronisation and communication in distributed systems, transactions, persistency and security concepts. The underlying complexity of those solutions is hidden from developers, who thus are able to focus on the business logic rather than on low-level and infrastructure functionality. We believe that a similar approach concerning hiding the complexity of real-time robot control can substantially improve the speed, efficiency and quality of robotic software in the future.

III. Architecture

To achieve the desired abstraction of real-time critical tasks, a three-tiered architecture, as shown in Figure 1, was chosen. The *Robotics Application Programming Interface (RAPI)* offers the application developers an object oriented interface for robot programming. The RAPI is implemented by the *Robotics Base Class Library (RBCL)* in C#. Library developers are able to extend the RAPI with own interfaces (and implementations). These extensions can combine basic functionality to add convenient features for domain-specific tasks or add support for new hardware (e.g. special sensors or tools). An example for a function provided by the RAPI is a motion. The *glueing* instruction, which is used as an example in Section IV, is a domain-specific extension of the RAPI.

An easy-to-use interface like RAPI alone does not solve the problems concerning real-time issues in robotics. Although the RBCL can take care of many scheduling and synchronisation issues, it cannot provide means for the execution of real-time critical parts of the programs, because RBCL itself is still a library running in a non real-time environment. *The Realtime Primitives Interface (RPI)* offers

a descriptive language for issuing real-time critical tasks to a robot controller. The *Robot Control Core (RCC)* is a special part of a robot controlling system that implements the RPI. The RCC is the only part of the architecture that must be implemented with real-time aspects in mind. The RCC must be executed within a real-time operating system such as RTAI or VxWorks.

In the following sections, the different layers and interfaces are described in more detail, starting from the bottom with the Realtime Primitives Interface and the Robot Control Core.

A. Realtime Primitives Interface

As stated in Section II, high-level programs often contain only a small set of real-time critical tasks. The Realtime Primitives Interface (RPI) is an interface for submitting such tasks to the Robot Control Core. All tasks that have been submitted using RPI will be executed atomically under real-time conditions. It is important to note that RPI is not an interface as known from object-oriented programming languages. The RPI is a language that describes a real-time critical task in a formal way. Instead of executable code, only data describing a task is transferred between the RBCL and the RCC. Tasks expressed with RPI consist of two main concepts:

1) *Realtime primitives (RT-primitives)* are functions that map n input values to m output values where $n, m \in \mathbb{N}_0$. In particular, RT-primitives with no input or no output values are legitimate. Examples for basic RT-primitives are robot joint controls or trajectory planners.
2) A *link* connects an output value of one RT-primitive with an input value of another RT-primitive. A link is always connected with exactly one RT-primitive on the output side, and another RT-primitive on the input side. An output value may have multiple links attached, whereas an input value may only be connected to a single link.

Furthermore, new RT-primitives can be created by composing existing RT-primitives and links. Composed RT-primitives are a convenience feature and allow easy reuse of already existing functionality. When transmitted to the RCC, all composed RT-primitives can be unfolded to contain only atomic RT-primitives and links. The example in Section IV makes use of composed RT-primitives.

RT-primitives and links form a graph. RT-commands are special graphs which are acyclic and do not contain RT-primitives with any unconnected input or output value. The execution of RT-commands is done periodically on the Robot Control Core. In each period, the whole graph is evaluated. This means that every RT-primitive has read all its input values, potentially performed a calculation and written all its output values. Because the graph is acyclic, a complete linearisation is possible, providing a sequential order of RT-primitives for execution. This execution semantic is closely related to the execution semantics of widely used tools like Matlab Simulink [4] or Scade [5]. The *Worst Case Execution Time (WCET)* of the whole RT-command is the sum of the WCETs of all RT-primitives plus a small offset for the time needed to transfer data among the primitives. Naturally, the period duration for evaluating the RT-command must be at least as big as its WCET.

There is no exhaustive list of RT-primitives. A few basic RT-primitives are sufficient for using a standard industrial robot in standard tasks. However, special actuators, sensors etc. will need to define new RT-primitives and provide a real-time capable implementation.

B. Robot Control Core

The Robot Control Core (RCC) is an interpreter for RT-commands under real-time conditions. It is responsible for direct hardware control. It needs the ability to communicate with the robot hardware, e.g. define the set points for the servo motors. The core must provide all functionality that is real-time critical and, thus, is usually running on a real-time operating system. Real-time critical actions include the actual robot movement (new set points must be defined at exact intervals to ensure a smooth and deterministic movement) and the control of sensors and tools (e.g. a welding torch must be turned on and off at exactly the right positions).

Implementation details of the Robot Control Core are not of interest for the architecture, as long as the RCC can execute tasks specified with RPI. This allows for replacing the core. For example, an application can be programmed based on a simulated robot control and subsequently transferred to a *real* robot control.

C. Robotics Application Programming Interface

The Robotics Application Programming Interface (RAPI) provides means for the application developer to program a robot. It is object-oriented and offers a set of classes and routines in order to support the building of robotic applications. Main concepts of the RAPI are objects for robots, tools, frames or instructions (e.g. motions or tool actions). Developers are able to extend the RAPI in order to encapsulate domain-specific functionality or add new robots and tools.

Although the RAPI hides real-time issues, some synchronisation is necessary in robotics. Therefore, the RAPI provides special operators that are able to combine multiple instructions. For example, there are operators especially for combining instructions with a defined temporal relation. Further common operators (e.g. superposition) are also available.

D. Robotics Base Class Library

The Robotics Base Class Library (RBCL) is an implementation of the RAPI. The RBCL must provide means to retrieve objects for all available robots and tools. Furthermore, common instructions for different types of motion must be provided. The RBCL must map the high-level objects like robots, tools and instructions to low-level RT-commands and RT-primitives. The synchronization hereby is solved with the deterministic, periodic evaluation of RT-commands.

IV. Example

To illustrate the approach, a small example is presented showing how a robot instruction, specified in C#, can be translated into an RT-command that can be executed within a real-time environment on the Robot Control Core. In the example, the task is glueing a straight line on a work piece. In practice, starting the glueing gun does not immediately start the output of glue because some hoses might need to be filled first and a certain pressure at the nozzle must be reached. Therefore, the glueing gun must be started slightly previous to the motion.

During the analysis and design phase, the application developer decided to create a class `Glueing` that encapsulates all required functionality for the glueing task. Having created this class once, he can reuse the functionality many times without having to think about the glueing details again. The resulting class diagram of the object-oriented design is shown in Figure 2. The `Glueing` class needs associations with a robot and a glueing gun, and supports setting an additional filter.

Internally, the implementation of the `Glueing` class must use basic objects from the RAPI library like robots, tools, instructions and operators. The glueing task can be separated into two fundamental tasks. One task is the motion of the robot (more exactly, of the tool centre point), the other the control of the glueing gun. Although both instructions can be executed independently, for a correct result it is indispensable that both actions are timed highly accurate. The gun may be turned on neither too early, nor too late. The RAPI offers means to synchronize two instructions with a specified temporal relation. For the glueing example, the gun must be turned on immediately, while the linear movement must be delayed slightly (but for a precise time).

The implementation of the `Glueing` class for executing the task is shown in the following listing.

```
public class Glueing {

  public void Execute() {
    LinearPath path =
      new LinearPath(this.startPoint, this.stopPoint);

    Instruction motion =
      new MotionInstruction(path, this.actuator);
    Instruction guncontrol =
      new GunControlInstruction(this.gun);

    Instruction synchronize =
      new Synchronize(this.preStart, guncontrol, motion);

    // execution omitted
    ...
  }
}
```

Listing 1. Simplified implementation of the method `Execute` in the `Glueing` class.

The `Execute` method contains only calls to RAPI functions. At the beginning, a new `LinearPath` object is instantiated with values for the start and destination points. The next step is the creation of a `MotionInstruction` object, which connects the path with a robot that executes the motion. Independently of the motion, a `GunControlInstruction` object is instantiated. Subsequently, both instructions need

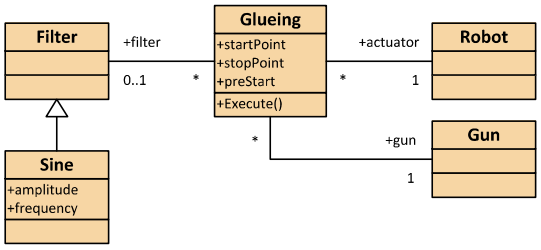

Fig. 2. UML class diagram modelling the functional requirements of the glueing task.

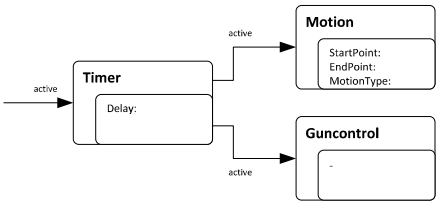

Fig. 3. Composed RT-primitive from mapping the synchronized motion and gun control instructions.

to be synchronized. This is done using the `Synchronize` operator, which delays the start of the motion instruction. Finally, the generated object structure needs to be converted to an RT-primitive which can be executed. For this purpose, a mapping mechanism is employed which recursively traverses the object structure and tries to map each part to its representation in RPI. Therefore, each object needs to provide information about its mapping. For example, the `LinearPath` and its parametrisation is mapped to a RT-primitive called *TrajectoryPlanner*.

Figure 3 shows the RT-primitive obtained by the mapping in a graphical way. Each block represents one RT-primitive. The name of the RT-primitive is displayed in the upper left corner, the time-invariant parameters in the box in the lower right corner. Input values are displayed on the left side of a block, output values on the right side. The `Synchronize` operator is mapped to a *Timer* block that handles the timing issues, and controls the RT-primitives *Motion* and *Guncontrol* with the *active* port they provide. The *Motion* primitive is the result of mapping the `MotionInstruction` and consists of a *TrajectoryPlanner* (from mapping the `LinearPath`) and a *Robot* primitive, as it can be seen in Figure 4.

In the glueing example, the synchronisation has been done using a fixed time delay. It is also possible to trigger actions on geometrical conditions, i.e. when a certain point is reached. Basically there are two ways of achieving a geometrical trigger. (1) If no real-time sensor value is considered, the planned trajectory is deterministic which allows to convert the geometrical position into a time constraint again. (2) It is also possible to design a synchronisation module that reads the trajectory position and triggers an action depending on the current actual position which can be be sensor-corrected.

Sometimes, it is not only desired to glue along a straight line, but slightly change the position of the gun in a sine wave over the work piece to enlarge the area covered with

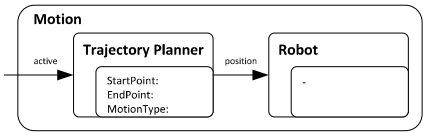

Fig. 4. Composed RT-primitive from mapping the motion instruction. It consists of a trajectory planner for the linear path and a primitive representing the robot.

glue. The `Glueing` class might provide a special setting `Filter` to overlay the linear path with another function. An excerpt of the modified method `Execute` is displayed in the following listing.

```
public class Glueing {

  public void Execute() {
    LinearPath path =
      new LinearPath(this.startPoint, this.stopPoint);

    if(this.filter != null)
      path = new Superposition(path, this.filter);

    Instruction motion =
      new MotionInstruction(path, this.actuator);

    ...
  }
}
```

Listing 2. Modified `Execute` method to support additional filters which modify the trajectory.

If the instance variable `filter` is available, the linear path is first modified with the operator `Superposition`. The result of applying this operator can be used in the creation of the motion instruction exactly like the original linear path. The synchronization does not need to be changed. The Timer primitive in Figure 3 now simply uses the more complex motion primitive of Figure 5 instead of the primitive of Figure 4.

Once the `Glueing` class has been defined, it can be easily used in new C# programs. Listing 3 displays the usage of the class with a sine wave superposed to the linear trajectory. The developer now does not need to care about real-time, or even synchronization of multiple instructions.

```
public void DoSineGlueing(...) {
  Glueing glue = new Glueing();
  glue.startPoint = startPoint;
  glue.stopPoint = stopPoint;
  glue.preStart = start;
  glue.actuator = robot;
  glue.gun = gun;
  Sine sine = new Sine(amplitude, frequency);
  glue.filter = sine;
  glue.Execute();
}
```

Listing 3. Exemplary program for glueing along a straight line, superposed with a sine wave.

V. Related Work

Robot programming languages are specifically developed programming languages with robot-specific instructions and commands derived from early imperative languages like ALGOL or Pascal. Typically, they are provided by robot manufacturers and can be used to create simple programs for their robot controllers. Examples for these languages

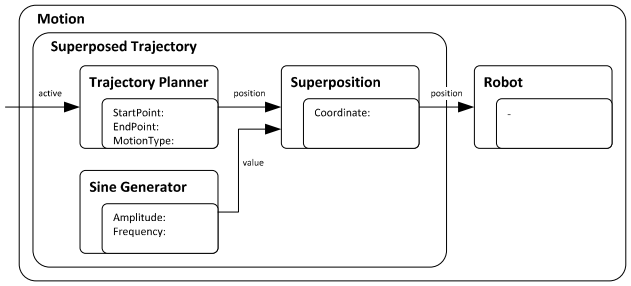

Fig. 5. Composed RT-primitive from mapping the motion instruction with superposed sine. It was generated by modifying the trajectory planner with the superposition operator.

are VAL-II [6], the KUKA Robot Language, ABB's robot programming language RAPID, or KAREL developed by FANUC Robotics. Although these languages hide real-time issues from the application developers, they are still proprietary, low-level languages and it is difficult to use libraries written in general-purpose languages (e.g. for graphical user interfaces). To overcome these limitations, robot manufacturers have started to provide libraries for general-purpose languages. These libraries are based on the underlying controller-specific language and can be seen as simple wrappers. Hence, they do not offer the possibility to extend the library and dynamically generate new commands. An example is the Robot Application Builder from ABB providing a programming interface in C#.

Furthermore, there are several academic approaches providing robot-specific libraries for general-purpose languages. Early examples are RCCL [7] in C and PasRo [8] in Pascal. With the emergence of object-oriented languages, robot-specific libraries for these languages have been implemented, too. Examples are ZERO++ [9] and MRROC++ [10]. These C++ libraries are designed for real-time operating systems and, as a consequence, do not hide real-time issues from the application developer. A robot command library using the imperative language Occam is presented in [11]. This library is based on two proprietary robot controls and provides eight different commands which are an intersection of the two underlying robot controls. Pires and da Costa [12] use distributed objects to remotely call predefined operations of the robot control. The Robotic Platform [13] also provides an object-oriented model for programming robots, but does not abstract from real-time issues.

Besides, there have been major efforts in the research community to develop modular robot control frameworks. An open source object-oriented framework for controlling robots is OROCOS [14]. It is written in C++ and allows for developing component-based real-time control applications. Moreover, it provides ready-to-use components (e.g. for kinematics, dynamics, and motion control). Other real-time capable frameworks are e.g. Player [15], Orca [16], MiRPA [17], and SIMOO-RT [18]. These frameworks have in common that they focus on real-time robot control and use object-oriented concepts. In contrast to our approach, the abstraction of real-time issues and the mapping from

high-level commands to low-level robot control is secondary. However, these frameworks could be used to implement the Realtime Primitives Interface. A real-time programming environment for robotic systems written in C++ is ORCCAD [19]. It introduces a three-layered architecture to offer different abstraction levels (from control to application tasks) and uses the Esterel language to specify robot control laws as well as application logic.

The manipulation primitives approach [20] is similar to RPI where a complex robot task is also segmented into single motions and tool actions. However, in RPI the atomic items (RT-primitives) are more generic, as they only describe parts of a motion or a tool action and can be combined to form any kind of command.

VI. Conclusion

We have proposed a three-tier software architecture for programming and controlling industrial robots. By using standard technologies, this architecture enables the application of modern methods of software engineering. However, the main problem that must be addressed is the separation of non real-time application logic and real-time critical control of actuators. The core idea is that robotic applications can be split into small subtasks requiring hard real-time constraints. The orchestration of these subtasks, i.e. the application logic, can be performed inside a non real-time environment.

In order to submit such real-time critical subtasks to a robot controller, we have introduced an extensible, descriptive language, the Realtime Primitives Interface. On top of this interface, we have outlined the Robotics Application Programming Interface. This interface provides an object-oriented approach to robot programming. Main concepts are objects representing robots, tools, sensors and instructions which application developers can use. The RAPI also transparently handles the mapping to low-level, real-time primitives, providing high-level support for commands synchronizing multiple robots and tools. By using this interface, application developers can fully concentrate on the domain-specific problems and do not have to worry about low-level real-time issues. From our point of view, this approach enables efficient, fast and affordable software development for industrial robots.

Furthermore, our approach does not only work well for *standard* industrial applications like glueing or welding, but for every application which has a defined set of real-time critical tasks. Apart from the industrial domain, this approach can also be applied e.g. in service robotics. Because the Realtime Primitives Interface allows a very dynamic description of robot tasks, the integration of external input devices and sensors or the implementation of force/torque controlled motions are possible. Due to the RAPI and its mapping mechanism, such complex tasks can be implemented and used much more easily. Moreover, applications for cooperating robots or even mobile systems can be developed using the same techniques.

The approach has been successfully applied to a KUKA robot system. In order to show the advantages and the usability of our approach, next steps will focus on extending this prototype and realizing more complex examples. However, first results are very promising. In addition to that, we are currently working on an example application comparing our approach to programming in the KUKA robot language.

References

[1] U. Hagn, M. Nickl, S. Jörg, G. Passig, T. Bahls, A. Nothhelfer, F. Hacker, L. Le-Tien, A. Albu-Schäffer, R. Konietschke, M. Grebenstein, R. Warpup, R. Haslinger, M. Frommberger, and G. Hirzinger, "The DLR MIRO: A versatile lightweight robot for surgical applications," *Industrial Robot*, vol. 35, no. 4, pp. 324 – 336, 2008.

[2] I. Sommerville, *Software Engineering*, 8th ed. Addison-Wesley, 2007.

[3] J. G. Ge and X. G. Yin, "An object oriented robot programming approach in robot served plastic injection molding application," in *Robotic Welding, Intelligence and Automation*, ser. Lecture Notes in Control and Information Sciences, T.-J. Tarn, S.-B. Chen, and C. Zhou, Eds., vol. 362. Springer-Verlag, 2007, pp. 91–97.

[4] MATLAB Simulink. The MathWorks. [Online]. Available: http://www.mathworks.com/products/simulink/

[5] SCADE Suite. Esterel Technologies. [Online]. Available: http://www.esterel-technologies.com/products/scade-suite/

[6] B. E. Shimano, C. C. Geschke, and C. H. Spalding, "VAL-II: A new robot control system for automatic manufacturing," in *Proceedings of the 1984 IEEE International Conference on Robotics and Automation*, March 1984, pp. 278–292.

[7] V. Hayward and R. P. Paul, "Robot manipulator control under unix RCCL: A robot control C library," *International Journal of Robotics Research*, vol. 5, no. 4, pp. 94–111, 1986.

[8] C. Blume and W. Jakob, *Programming Languages for Industrial Robots*. Springer-Verlag, 1986.

[9] C. Pelich and F. M. Wahl, "ZERO++: An OOP environment for multiprocessor robot control," *International Journal of Robotics and Automation*, vol. 12, no. 2, pp. 49–57, 1997.

[10] C. Zieliński, "Object-oriented robot programming," *Robotica*, vol. 15, no. 1, pp. 41–48, 1997.

[11] G. Tewkesbury and D. Sanders, "A new robot command library which includes simulation," *Industrial Robot*, vol. 26, no. 1, pp. 39–48, 1999.

[12] J. N. Pires and J. S. da Costa, "Object-oriented and distributed approach for programming robotic manufacuring cells," *IFAC Journal on Robotics and Computer Integrated Manufacturing*, vol. 16, no. 1, pp. 29–42, 2000.

[13] M. S. Loffler, V. Chitrakaran, and D. M. Dawson, "Design and implementation of the Robotic Platform," *Journal of Intelligent and Robotic System*, vol. 39, pp. 105–129, 2004.

[14] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, Seoul, Korea, May 2001, pp. 2523–2528.

[15] T. H. Collett, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proceedings of the 2005 Australasian Conference on Robotics and Automation*, C. Sammut, Ed., Sydney, Australia, December 2005.

[16] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, "Towards component-based robotics," in *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Edmonton, Canada, August 2005, pp. 163–168.

[17] B. Finkemeyer, T. Kröger, D. Kubus, M. Olschewski, and F. M. Wahl, "MiRPA: Middleware for robotic and process control applications," in *Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware. IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Diego, USA, October 2007, pp. 76–90.

[18] L. B. Becker and C. E. Pereira, "SIMOO-RT – An object oriented framework for the development of real-time industrial automation systems," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 421–430, August 2002.

[19] D. Simon, B. Espiau, E. Castillo, and K. Kapellos, "Computer-aided design of a generic robot controller handling reactivity and real-time control issues," *IEEE Transactions on Control Systems Technology*, vol. 1, no. 4, pp. 213–229, December 1993.

[20] B. Finkemeyer, T. Kröger, and F. M. Wahl, "Executing assembly tasks specified by manipulation primitive nets," *Advanced Robotics*, vol. 19, no. 5, pp. 591–611, 2005.