Service-oriented Robotics Manufacturing by reasoning about the Scene Graph of a Robotics Cell

Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein, Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg, Germany

Abstract

This paper presents a novel service-oriented approach to robotics manufacturing. The idea is that the automation software is composed of a hierarchy of services. In order to decide how to execute manufacturing tasks, the scene graph of a robotics cell (i.e. the physical objects, devices and spatial relations) is inspected to retrieve the current situational context of a manufacturing application. Based on that context, the best suitable strategy to accomplish a task is chosen. Hence, this context-aware service-oriented approach facilitates flexible robotics manufacturing.

1 Introduction

In robotics automation, flexible and adaptive manufacturing is getting more and more important. According to Pires [1], it can be considered as the most important requirement for increased use of industrial robot in the future. As product life-times and volumes are getting less predictable, the fast “adaption to new batches, product variants or new products” [2] is more and more important. As a consequence, the way of structuring programs for industrial robot has to change. At the moment, they usually consist of sequences of different motions between pre-defined positions. From our point of view, more information belongs to and should be attached to the objects that are manipulated [3]. Based on this information, the robot program has to decide how to process the object in order to accomplish its task.

To facilitate this novel paradigm of programming, we present an approach which smoothly integrates robot control (i.e. motion execution) and task description based on the objects to manipulate. This approach builds on an object-oriented framework for programming industrial robots with an application programming interface: the *Robotics API* [4]. Object-orientation allows for modeling the devices and physical objects that exist in a robotics cell as software objects and, thus, creating a scene graph of this cell. When executing operations on devices of the cell with hard real-time guarantees on the underlying *Robot Control Core* [5], the scene graph gets automatically updated. On top of this object-oriented framework, we present a service-oriented approach for reasoning about the robotics scene graph in order to decide how to manipulate a given object. The used of service-oriented paradigms can facilitate the flexibility and adaptability to changes that is required in manufacturing [6].

This paper gives an overview of the proposed approach by describing the robotics scene graph and by introducing the idea of context-aware service-oriented manufacturing. Section 2 gives an overview of the software architecture used for service-oriented manufacturing. In Section 3, we describe how the scene graph of a robotics cell is modeled and explain which information can be retrieved from this scene graph. Section 4 describes the idea of using the scene graph for service-oriented manufacturing in detail. Examples are given in Section 5 and related work in Section 6. Finally, Section 7 concludes the paper and gives an outlook.

2 Architecture

The presented framework is based on the software architecture (cf. **Figure 1**) developed in the joint research project *SoftRobot*. The framework is geared towards realizing complex, sensor-guided manipulation tasks of single robots or small teams of robots. The main idea [7] is that robotics software is developed against an application programming interface (API) – the Java-based *Robotics API* [4, 8] – and robot operations are executed with hard real-time guarantees on the underlying *Robot Control Core (RCC)*. Thus, the RCC is responsible for controlling hardware devices and must be implemented on a real-time operating system. This separation is possible, because industrial robotics applications do not inherently have real-time requirements for the application logic, but only for parts of the program like motions or tools actions. Hence, the RCC takes care of all real-time critical parts of the robotics systems and provides a flexible, generic interface: the *Realtime Primtives Interface* (RPI) which is described in detail by Vistein et al. [9]. RPI is a dataflow based language, consisting of basic *calculation modules*, which can be combined to form complex commands. Besides these basic calculation modules, there are also *device modules* for sending data to or retrieving data from sensors and actuators. Those drivers also need to be implemented in C++ for real-time requirements. The Robotics API automatically combines these calculation and device modules to create real-time commands [10] and submits them to the RCC for execution. Once the RCC has fully received such a command, it deterministically executes all contained modules, hence guaranteeing real-time characteristics.

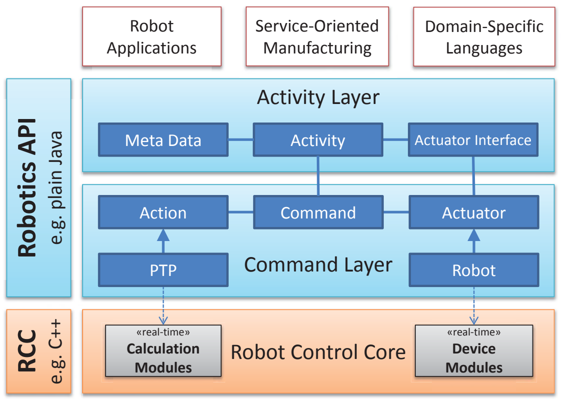
Robot Applications
Service-Oriented Manufacturing
Domain-Specific Languages
Robotics API e.g. plain Java
Activity Layer
Meta Data
Activity
Actuator Interface
Action
Command
Actuator
PTP
Command Layer
Robot
RCC e.g. C++
«real-time» Calculation Modules
Robot Control Core
«real-time» Device Modules

Figure 1: The software architecture for industrial robotics developed in the research project *SoftRobot*.

During execution, sensor drivers provide (raw or processed) sensor data at a high frequency. Calculation modules can further process such sensor data, calculate set-points for trajectories, or trigger tool actions. Finally, actuator drivers expect to receive set-points (e.g. position, velocity, acceleration) at a high frequency (usually 1 kHz) from a running command. The reference implementation – the *SoftRobot RCC* – is implemented in C++ and is targeting Linux Xenomai for robot control as well as Windows for testing and simulation purposes.

The Robotics API consists of two layers: the *Command Layer*, which directly communicates with the Robot Control Core, and an additional *Activity Layer* which provides more convenient access to the most common functions of the Robotics API. The Command Layer has a very abstract model of robotic tasks, which consists of actions which can be performed by actuators. Assigning an action to an actuator yields a command, which can be executed using a Robot Control Core. Actions can be for example a Point-to-Point (PTP) motion, and an actuator can be a specific type of robot arm. Actuators need a corresponding driver and primitives in the Robot Control Core, whereas actions usually map to a set of calculation modules which together form the requested action [11]. Using an event mechanism, multiple commands can be combined e.g. to perform tool operations based on sensor data. Furthermore, the Command Layer gives a transparent access to real-time data (e.g. joint positions, sensor data) which is obtained from the RCC. It provides a very generic, flexible and extensible interface for writing arbitrarily complex applications. Hence, it is mainly designed to integrate new devices, motion types etc. into the system. In contrast, the Activity Layer provides an intuitive execution model for commands. It wraps commonly used functions into Java methods and is therefore targeted at robot-application developers. More details about these layers can be found at Angerer et al. [4].

3 Robotics Scene Graph

Robotic applications always need to describe some part of the physical world or environment the robot is situated in. In standard robot programming languages (e.g. the KUKA Robot Language, Staubli's VAL), one has to define at least points in space that are necessary for the specification of robot motions. Depending on the concrete application, more information about the environment is desired. For example, in assembly applications, the notion of workpieces that have to be assembled can be helpful. However, such complex models are predominantly supported by offline programming tools, whereas standard robot controls usually only support the definition of points. The scope of the Robotics API comprises both use cases, because it supports the definition and execution of (real-time) robot commands, as described previously, and the complex modeling of robotics cells. Therefore, the definition of spatial points, geometric relations and physical objects is possible as shown in **Figure 2**. The figure shows a (simplified) excerpt from the Robotics API object-oriented programming model using an UML class diagram. The left part of the diagram which contains frames and relations is described in Section 3.1, whereas the right part – i.e. the modeling of physical objects and devices – is explained in Section 3.2.

3.1 Frame & Relations

The scene graph of the Robotics API is based on a graph of inter-connected frames. *Frames* are (named) positions in space which can optionally belong to (the software representation of) a device or in general a physical object. They are interconnected by relations which have a defined semantics. As shown in Figure 2, a *Relation* connects a pair of Frames, specifies the kind of connection between those two frames, and defines the geometric transformation between them. The *Transformation* is mathematically expressed by a homogeneous transformation matrix. Instead of having a parent-child-relationship between frames, a graph is formed where frames are vertices and relations are edges. Hence, each frame can have an arbitrary number of relations directing to or originating from it. A frame graph is called connected if and only if there is at least one path between any two frames of the graph. In a connected frame graph the transformation between any two frames can be determined. However, it is possible to have multiple independent frame graphs at a time. During the execution of an application, frame graphs can be joined or separated by adding or removing relations.

The *Relation* class itself is abstract, but there are several subtypes with different semantics describing the kind of

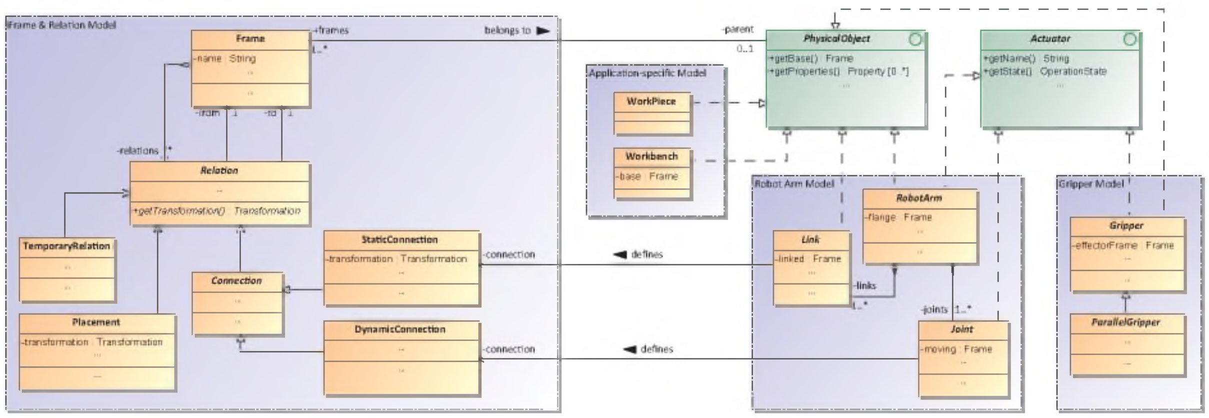

Figure 2: UML class diagram showing important parts of Robotics API class hierarchy for modeling the scene graph of a robotic cell

relation between two frames. For expressing the importance and permanence of a relation, there is a distinction between a temporary relation, a placement and a connection. A *TemporaryRelation* is a transient relation which is only used to create auxiliary frames for motions or calculations. It is designed in such a way that it is automatically removed when it is no longer used in an application. A *Placement* indicates that - at least at the moment – two frames are connected. Similar to a temporary relation, this is also only a snapshot and the placement can be changed or removed during the execution of the application. However, it is a persistent relation (i.e. the relation must be removed on purpose) and, as its name implies, is used e.g. to indicate that some object is currently placed on a table. In contrast, *Connections* are long-lasting relations between frames. For example, a link defines a (mechanical) connection between two joints which usually will not be removed. Furthermore, there is a distinction between static and dynamic connections. While a static relation defines a fixed transformation (e.g. a given displacement between two objects), the transformation of a dynamic relation can change over time as it is usually defined by an actuator. For example, a revolute joint consists of two frames which are rotated against each other according to the joint's current position. The transformation of dynamic relations is automatically provided by the framework. For this purpose, the dynamic connection internally uses sensors provided by the Robotics API which communicate transparently with the RCC (c.f [4, 10]). Developers can create own subtypes of *Relation* to express a application-specific connection between two frames (e.g. a grasping placement).

3.2 Physical Objects & Devices

Physical objects are software objects that have a counterpart in the real cell. They can be either *passive* objects such as work pieces or *active* objects i.e. actuators such as robot arms or tools. Moreover, physical objects can consist of multiple parts. Figure 2 shows, for example, a robot arm which consists of links and joints. However, their spatial relations are defined through frames that belong to these physical objects. Joints define a dynamic connection between their two frames. By convention, this dynamic connections rotates the joint's moving frame around the z-axis of the joint's base frame according to the current joint position. This value is transparently retrieved from the RCC and updates the transformation. Links define a static connection from their base frames to their linked frames according to their dimensions. Moreover, different physical objects can be arranged using static connections to build up a robotics cell.

Figure 3 illustrates parts of an exemplary robotics cell using an UML object diagram. It shows a robot arm with its last two links and the joint between those links. The depicted joint shares its frame with the two links. These frames are associated with static and dynamic connections in alternating order. Hence, this definition of links and joints forms the robot's shape. Furthermore, the figure shows that the robot arm is mounted on a workbench using a static connection between the base frame of the workbench and its own base frame. The static connection between the robot's flange and the gripper's base frame implies that the gripper is mounted on the robot's flange.

Because frames and relations can belong to physical objects, the frame graph can be inspected to draw conclusions about the robotics scene. For example, it is possible to calculate a path of frames and relations between physical objects. By having semantically enriched relations between frames, it is possible to determine the kind of relation between physical objects. Based on a path of frames and relations between physical objects, it is possible to retrieve the relevant actuators connecting these objects. In order to ensure a consistent scene graph, relations must be manipulated with care. For example, after grasping an object that is placed on a table, the placement must be removed and a new relation to the gripper must be established. By doing so, the framework will automatically update the object's position with respect to the table when the gripper is moved.

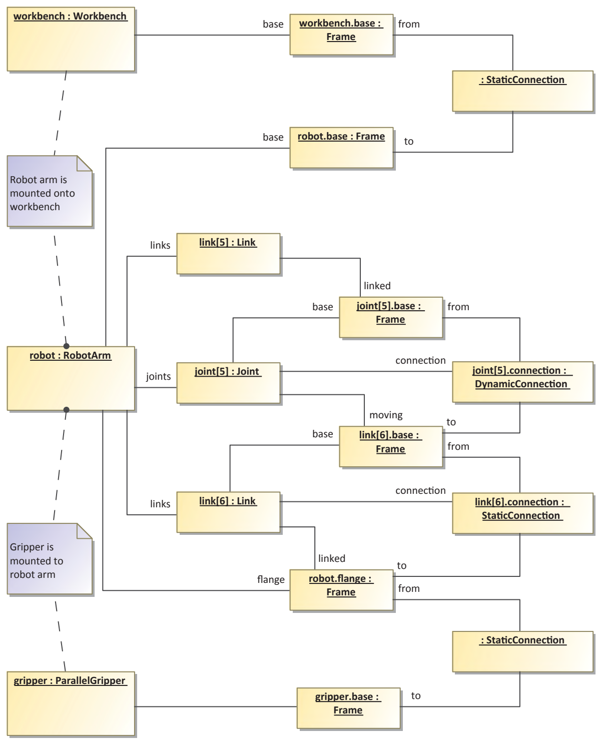

Figure 3: Excerpt from an UML object diagram showing a robot arm mounted on a workbench. A parallel gripper is mounted at the flange of the robot.

4 Services for Robotics Tasks

This section describes the approach of using context-aware services for robotics manufacturing. The (situational) context is based on the input parameters of the service (i.e. actuators and physical objects) and retrieved using the current scene graph. The main idea of this approach is described in Section 4.1. Details about the realisation are given in Section 4.2.

4.1 Approach

The idea of service-oriented manufacturing is to compose the automation software of a hierarchy of services on different levels of abstraction. Each service has its responsibility in the automation system, but can assign (sub)tasks to other services which are usually on a lower abstraction level. The parameters for automation services are the physical objects that are processed or manipulated by the automation system. In order to execute robot or tool operations, we introduce task-level services. Such a service describes a particular task using appropriate parameters (e.g. grasping an *object* with a *gripper*, assembling two *objects*, or placing an *object* into another *object*). For the execution of a task, the service inspects the current scene graph and retrieves the situational context of tasks. Based on this context, an appropriate strategy for this task is chosen, parametrized and executed.

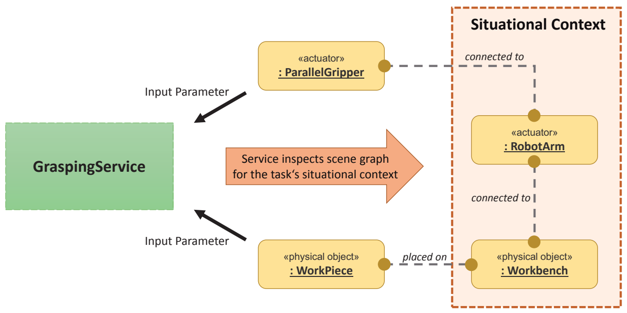
Situational Context
«actuator» : ParallelGripper
connected to
Input Parameter
«actuator» : RobotArm
GraspingService
Service inspects scene graph for the task's situational context
connected to
Input Parameter
«physical object» : WorkPiece
placed on
«physical object» : Workbench

Figure 4: A grasping service retrieves the situational context using its input parameters (i.e. the gripper and the work piece) and searching the current scene graph.

Our approach is shown in **Figure 4** using a *GraspingService* as example. The input parameters of the service are the work piece to grasp and the parallel gripper. The structure is similar to the example in Figure 3, i.e. the gripper is mounted on a robot arm which is mounted on a workbench. The work piece is simply placed on the table. The situational context can be retrieved completely from the scene graph as described in Section 3. This is important because some actuators, in particular robot arms, are not part of the task description. However, they support the task by moving tools and objects. This approach covers many scenarios such as an externally mounted tool, two objects both carried by different robot arms or one object cooperatively carried by multiple robot arms. As shown in the example, the scene graph is also used to retrieve more information about the objects to manipulate, e.g. whether the object is placed on a table, in a box, or in a fixture. Hence, the scene graph enables the service to retrieve the task's situational context.

Being able to retrieve the context, the task-level service can select a strategy to execute the operations for successfully accomplishing the task. In accordance to the *Strategy* pattern introduced by Gamma et al. [12], a task strategy is a solution for a characteristic *problem*. For each family of problems (e.g. grasping), several solutions – i.e. the strategies – can be applied that only share a common interface and, thus, are interchangeable to a certain extend. The main feature of this approach is that the best suitable strategy is chosen based on the situational context retrieved from the robotics scene graph.

For accomplishing a task, there are different possible strategies. For grasping a workpiece, a simple strategy is to approach from a given direction to a given (taught) position and close the gripper. However, there are also more advanced solutions using a compliant robot arm. In both cases, the strategy must be parametrized in order to work properly. These strategy parameters (e.g. tolerances, desired forces) can be easily attached to the software representation of physical objects and will be selected based on the situational context from the task-level service. It is also possible to have specialized strategies which are only suitable for one particular situation (described by the situational context).

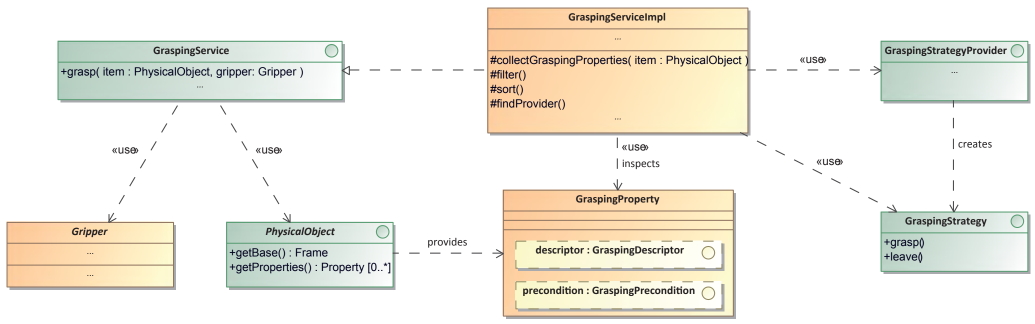

Figure 5: UML class diagram showing the structure of a *GraspingService*.

4.2 Realization

The context-aware task-level services are implemented in Java using the Robotics API and OSGi. The OSGi framework is a dynamic module system and service platform for Java. Since 1999, its specification [13] has been developed by the members of the OSGi Alliance (e.g. IBM, Oracle, Samsung, Siemens). An application based on OSGi is composed of many different (reusable) components called *bundles*. As in plain Java, bundles are packaged and deployed as JAR files. OSGi implements a service model and introduces a *service registry*. Using this registry, bundles can register or retrieve services, and they can listen for services to appear or disappear. Similar to bundles, services are also dynamic, i.e. a bundle can decide to withdraw its services from the registry.

The implementation of task-level services is explained by taking the example of a *GraspingService* (cf. **Figure 5**). The service is described using an interface which decouples the use of the service from different possible implementations. As mentioned above, the input parameters for grasping are a gripper and the item to grasp. The item is a physical objects which may provide *GraspingProperties*. Such a property consists of a *precondition* and a strategy *descriptor*. The descriptor corresponds to a specific strategy and contains the relevant parameters to instantiate the strategy. The precondition is application-specific and evaluates to true if and only if the corresponding strategy can be executed in the current situational context. For this purpose, the precondition can inspect the current scene graph starting from gripper or the item.

To instantiate task strategies, the implementation of the *GraspingService* uses strategy providers. These providers are registered at the service implementation and can then be used. When the grasp operation is invoked at the service, the implementation starts by collecting all grasping properties from the given item. By evaluating the application-specific precondition, the grasping properties are filtered to obtain only suitable strategy descriptors. The set of suitable strategy descriptors is sorted using a application-specific priority. For the best strategy descriptor, the service retrieves the appropriate strategy provider. If no provider is found, the service continues with next strategy descriptor. Before instantiating the strategy, the provider inspects the current scene graph and evaluates if the situational context is suitable to its strategy. Hence, the provider is able to ensure that all requirements are meet to successfully execute the strategy. If the provider's precondition evaluates to false, the service continues with next strategy descriptor. Then, the provider instantiates a new strategy using the parameters from the given descriptor. Finally, the service implementation starts to execute the strategy and monitors its execution.

5 Case Study: Factory 2020

To illustrate service-oriented manufacturing, a robotic assembly application called Factory 2020 was developed. The application was designed along the vision of a fully automated factory of the future, in which different kinds of robots cooperate to perform complex tasks. In the demonstrator, two robot arms and a mobile robot platform work together in part assembling. To achieve that, service-oriented manufacturing uses force-based motions, real-time motion synchronization and different coordination patterns.

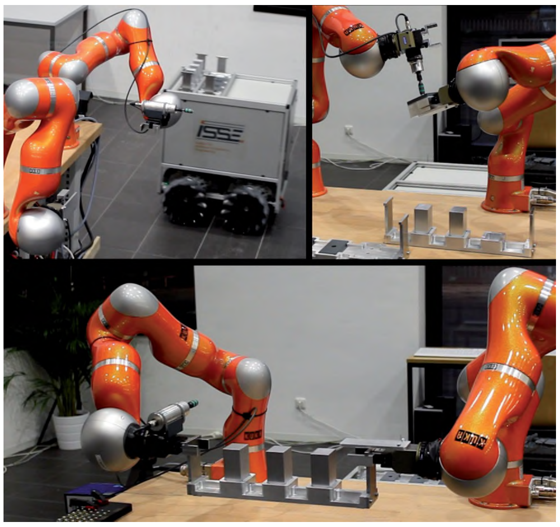

Figure 6: In Factory 2020, an autonomous robot platform delivers parts (top left), robot arms cooperate in transporting (bottom) and assembling them (top right).

Figure 6 gives an overview about the tasks performed. Two different containers with parts to be assembled are delivered by an autonomously navigating robot platform. Before assembling, both containers have to be transported onto the workbench. The robot arms locate the exact position of the containers by touching certain prominent points. This is required because the position may vary due to inaccurate navigation of the platform and some inaccuracy of the containers on the platform. After locating, both arms grip the containers and cooperatively transport them onto the workbench. These motions have to be real-time synchronized to ensure proper transport without damaging the containers or robots.

Once the containers have been placed on the table, each robot arm picks a part from one of the containers. To assemble the parts, the robots apply a defined force on the parts to compensate slight variations in the fitting between top and bottom part. Finally, both parts of the final workpiece have to be bolted together. For this purpose, an electrical screwdriver is attached to one of the robots as its second tool. This robot first fetches a screw from a screw magazine, then transports it to the workpiece, inserts it and tightens the screw using the screwdriver. These operations are also performed using force-based motions, which again allows for compensating variations in part quality and the process itself (e.g. slight deviations when gripping a workpiece part). The final workpiece is then put back into one of the containers. After all workpieces have been assembled, the containers are put back onto the platform, which delivers the workpieces to their destination.

For structuring the complete automation system software, it was divided into several services and components. Service-orientation eased deployment of components to different physical systems (robot arms, mobile base) and the coordination of those systems. For implementing the components which coordinate the interaction with and among other components, state machines were employed. The standardized State Chart XML [14] format was used to formally model those state machines. Based on the Commons SCXML [15] implementation, a graphical editor and a runtime environment for SCXML state machines was created.

On the lowest service layer, different task-level services were employed to control the robots and the tools. They were successfully used for tasks such as grasping or placing workpieces, for assembly, and to fasten the workpieces. For the first iteration of the software, for every task a unique strategy was implemented, tested and used. In later iterations, these strategies have been generalized and can be used in different situation with different parameters. These parameters have been attached to the work pieces as described in the previous section. It is possible to develop such strategies independently from the concrete application. Only the parameters have to be figured out in the context of the application. Reusing strategies eases on the one hand the development of similar applications and, on the other hand makes applications more robust.

6 Related Work

In [16], Smits makes a proposal for a *Robotics Scene Graph Standard* to have a standardized representation of robot systems. His proposal “covers the kinematic chain aspects of robot platforms, including the location of sensors, motors and tools on the chain, as well as the location of objects in the scene that are relevant to the robot’s task” [16]. He proposes an XML syntax based on COLLADA [17]. In contrast to our approach, he suggests to used a tree structure instead of a graph for frames. Furthermore, there are no semantics relations between frames. However, his standardization proposal could be realized with our approach. De Laet et al. [18, 19] present a set of concrete suggestions for standardizing terminology and notation which should allow “programmers to write fully unambiguous software interfaces” [18] with “semantic correctness of all geometric operations on rigid-body coordinate representations” [18]. ROS uses the Unified Robot Description Format [20] which assumes that a robot consists of rigid bodys connected by joints. However, it does not allow to specify relations from the robot to other physical objects of the environment. Blumenthal et al. [21] also present a scene graph based world model. Howwever, their focus lies on maintaining a shared world model for separate software components.

Veiga et al. [22, 23] use service-oriented platforms for industrial robotics. Their focus is on using service-oriented architectures – in particular UPnP – as a communication platform. In [24], a similar approach was presented by the authors of this paper for manufacturing carbon-fiber reinforced plastics. Naumann et al. present in [25] a control architecture for robot cells. Instead of directly programming devices, an interconnector provides a set of process commands. Considering to level of abstraction, process commands can be compared to task-level services. However, they do not incorporate the current state of the robot cell and attach manufacturing knowledge into the objects to manipulate. Schäfer and Lopez [26] propose to incorporate more knowledge about the manufacturing resources into the manufacturing control. From their point of view, this increases flexibility and manufacturing is more adaptive to changes. However, they do not explain how to exploit this knowledge to achieve this flexibility. Björkelund et al. [27] use a knowledge base using the *Resource Description Framework* [28] to improve skilled robot motions. Huckaby, Vassos and Christensen [29] developed a taxonomic framework for task modeling in manufacturing robotics. Based on this framework, a planner can calculate a sequence of skill to accomplish a more complex assembly task. However, execution still relies on some assumption, e.g. there are hardware-specific implementations for the lowest level of skills. Nonetheless, the approach could be an interesting complement to the our approach presented in this paper. Further approaches have been e.g. proposed by Kim et al. [30] and Simmons [31].

7 Conclusion

In this paper, we presented a novel service-oriented approach to robotics automation. By reasoning about the scene graph of a robotics cell, the best suitable strategy to accomplish a task is selected and executed. Using the Java-based OSGi framework and its service-oriented paradigm, the ideas presented here have been successfully implemented in a sophisticated manufacturing example: the Factory 2020 includes e.g. force-based assembly and cooperative transport with real-time motion synchronization. As the set of strategies is not fixed at run-time, the service-oriented automation software can even evolve over time, i.e. be adapted to new work-pieces, or be improved for existing ones. To prove the flexibility of service-oriented robotics manufacturing will be one of the next research topics (e.g. using planers). Furthermore, more research must be done on sensor-based relations that incorporate uncertainty. This leads to the questions of how to model this uncertainty in the scene graph and, of course, how to deal with it.

Acknowledgements

This work partly presents results of the research project *SoftRobot* which was funded by the European Union and the Bavarian government. The project was carried out together with KUKA Laboratories GmbH and MRK-Systeme GmbH and was kindly supported by VDI/VDE-IT GmbH.

References

[1] J. N. Pires, "New challenges for industrial robotic cell programming," *Industrial Robot*, vol. 36, no. 1, 2009.

[2] M. Hägele, T. Skordas, S. Sagert, R. Bischoff, T. Brogårdh, and M. Dresselhaus, "Industrial robot automation," European Robotics Network, White Paper, Jul. 2005.

[3] A. Angerer, A. Hoffmann, F. Ortmeier, M. Vistein, and W. Reif, "Object-centric programming: A new modeling paradigm for robotic applications," in *Proc. 2009 IEEE Intl. Conf. on Autom. and Logistics*, Shenyang, China, Aug. 2009.

[4] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "Robotics API: Object-oriented software development for industrial robots," *J. of Software Engineering for Robotics*, vol. 4, no. 1, pp. 1–22, 2013.

[5] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Flexible and continuous execution of real-time critical robotic tasks," *International Journal of Mechatronics and Automation*, vol. 4, no. 1, 2014.

[6] F. Jammes and H. Smit, "Service-oriented paradigms in industrial automation," *IEEE Trans. Ind. Inf.*, vol. 1, no. 1, pp. 62–70, 2005.

[7] A. Hoffmann, A. Angerer, F. Ortmeier, M. Vistein, and W. Reif, "Hiding real-time: A new approach for the software development of industrial robots," in *Proc. 2009 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, St. Louis, Missouri, USA*. IEEE, Oct. 2009, pp. 2108–2113.

[8] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "The Robotics API: An object-oriented framework for modeling industrial robotics applications," in *Proc. 2010 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, Taipeh, Taiwan*. IEEE, Oct. 2010, pp. 4036–4041.

[9] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Interfacing industrial robots using realtime primitives," in *Proc. 2010 IEEE Intl. Conf. on Autom. and Logistics, Hong Kong, China*. IEEE, Aug. 2010, pp. 468–473.

[10] A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif, "From robot commands to real-time robot control - transforming high-level robot commands into real-time dataflow graphs," in *Proc. 2012 Intl. Conf. on Inform. in Control, Autom. & Robot., Rome, Italy*, 2012.

[11] A. Hoffmann, A. Angerer, A. Schierl, M. Vistein, and W. Reif, "Managing extensibility and maintainability of industrial robotics software," in *Proc. 16th Intl. Conf. on Adv. Robotics, Montevideo, Uruguay*. IEEE, Nov. 2013.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Addison Wesley, 1994.

[13] *OSGi Core Release 5*, OSGi Alliance Spec., Mar. 2012. [Online]. Available: http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf

[14] *State Chart XML (SCXML): State Machine Notation for Control Abstraction*, W3C Working Draft 6, Dec. 2012. [Online]. Available: http://www.w3.org/TR/scxml/

[15] Commons SCXML. The Apache Software Foundation. [Online]. Available: http://commons.apache.org/proper/commons-scxml/

[16] R. Smits, "Robot skills: Design of a constraint-based methodology and software support," Ph.D. dissertation, KU Leuven, May 2010.

[17] M. Barnes and E. L. Finch. (2008) COLLADA - Digital Asset Schema Release 1.5.0. The Khronos Group. [Online]. Available: https://www.khronos.org/files/collada_spec_1_5.pdf

[18] T. D. Laet, S. Bellens, R. Smits, E. Aertbelien, H. Bruyninckx, and J. D. Schutter, "Geometric relations between rigid bodies (Part 1) – Semantics for standardization," *IEEE Robot. & Autom. Mag.*, pp. 84–93, Jun 2013.

[19] T. D. Laet, S. Bellens, H. Bruyninckx, and J. D. Schutter, "Geometric relations between rigid bodies (Part 2) – From semantics to software," *IEEE Robot. & Autom. Mag.*, pp. 91–102, Jun 2013.

[20] Unified Robot Description Format. Robot Operating System. [Online]. Available: http://wiki.ros.org/urdf

[21] S. Blumenthal, H. Bruyninckx, W. Nowak, and E. Prassler, "A scene graph based shared 3D world model for robotic applications," in *Proc. 2013 IEEE Intl. Conf. on Robot. & Autom., Karlsruhe, Germany*, 2013, pp. 453–460.

[22] G. Veiga, J. Norberto, and K. Nilsson, "On the use of service oriented software platforms for industrial robotic cells," in *Proc. IFAC International Workshop Intelligent Manufacturing Systems*, Alicante, Spain, May 2007.

[23] G. Veiga, J. N. Pires, and K. Nilsson, "Experiments with service-oriented architectures for industrial robotic cells programming," *Robotics and Computer-Integrated Manufacturing*, vol. 25, no. 4-5, pp. 746–755, 2009.

[24] A. Angerer, C. Ehinger, A. Hoffmann, W. Reif, G. Reinhart, and G. Strasser, "Automated cutting and handling of carbon fiber fabrics in aerospace industries," in *Proc. 6th IEEE Conf. on Autom. Science and Engineering (CASE 2010), Toronto, Canada*. IEEE, Aug. 2010, pp. 861–866.

[25] M. Naumann, K. Wegener, and R. D. Schraft, "Control architecture for robot cells to enable plug'n'produce," in *Proc. 2007 IEEE Intl. Conf. on Robot. & Autom.*, Rome, Italy, Apr. 2007, pp. 287–292.

[26] C. Schäfer and O. Lopez, "An object-oriented robot model and its integration into flexible manufacturing systems," in *Multiple Approaches to Intelligent Systems*, ser. LNCS. Springer, 1999, vol. 1611, pp. 820–829.

[27] A. Björkelund, L. Edström, M. Haage, J. Malec, K. Nilsson, P. Nugues, S. Gestegård Robertz, D. Störkle, A. Blomdell, R. Johansson, M. Linderoth, A. Nilsson, A. Robertsson, A. Stolt, and H. Bruyninckx, "On the integration of skilled robot motions for productivity in manufacturing," in *Proc. 2011 IEEE/CIRP Intl. Symp. on Assembly and Manufacturing, Tampere, Finland*, 2011.

[28] *RDF Schema*, W3C Std., Rev. 1.1, Feb. 2014. [Online]. Available: http://www.w3.org/TR/rdf-schema/

[29] J. Huckaby, S. Vassos, and H. Christensen, "Planning with a task modeling framework in manufacturing robotics," in *Proc. 2013 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, Tokyo, Japan*, Nov 2013, pp. 5787–5794.

[30] G.-T. Kim, S. D. Cha, and D.-H. Bae, "Task.o object modeling approach for robot workcell programming," in *Proc. 21st Annual Intl. Computer Software and Applications Conference (COMPSAC '97)*, 1997, pp. 109–114.

[31] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *Proc. 1998 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, Victoria, Canada*, Oct. 1998.