A Virtual Robot Control Using a Service-Based Architecture and a Physics-Based Simulation Environment

Thomas Stumpfegger, Andreas Tremmel, Christian Tarragona, and Michael Haag

Abstract — Requirements for software used in robotic systems are usually quite different in comparison to regular commercial software. Due to the distributed nature of sensors and actors, the underlying software architecture needs to support this distributed structure. Also the close interaction of humans and machines calls for software that reacts quickly and reliably within this network. This paper presents how we used the Microsoft Robotics Studio (MSRS) as a platform to quickly acquire a ‘service oriented architecture’ (SOA) for a virtual robot control. The paper also discusses how using the MSRS SOA in robotic software and the use of the MSRS physical simulation help to obtain the required high quality standards.

I. INTRODUCTION

Creating robotic software for the industrial market is a challenging undertaking. Customers who buy industrial robots care especially about the robot – not the software that drives it. The software is usually regarded as a free addition to the robot.

This is in strong contrast to the elevated needs for various and often exotic features, which usually concern the integration of sensors, field bus systems or other specialized industrial equipment.

In the industrial robotic environment, the highest quality standard is needed for the safety requirement. The software has to ensure at any point in time, that humans working with or in the vicinity of robots cannot be harmed by them, which usually requires the usage of many connected sensors guarding the workspace.

This leads directly to the next stressed software requirement for robotic software – stability. Especially in safety critical passages robotic software has to ensure correct operation. Simply restarting an application like usually done for general office applications is not an option.

Simulation has to try to test as many of the possible inputs for the software as possible.

The robotic industry always needed a high level of quality for the software running its machines. A level of quality, which actually could be achieved with today’s software development tools, simulation tools and software architecture concepts as soon as we find a good common ground for the industrial market and the general market.

One sign of good engineering is to reuse modules that have proven themselves in their environment. This allows creating complex but stable applications in an ever shorter timeframe, instead of reinventing everything over and over again. To be able to use modules created by the large community of software developers, robotic engineers must be able to use standardized tools with standardized interfaces.

One barrier that prevents the usage of solutions and tools available in the software engineering community for usage in the robotics world is this missing common ground for often very specialized technologies. Communication in factories is still often done by field bus systems, if not bare IO. The move to mainstream technologies promises to benefit from the tremendous efforts invested by the PC and gaming industry. Modern software development environments and software architectures which have reached a certain stage of maturity can be applied also in the robotics domain.

For these reasons we are researching the recently released Microsoft Robotics Studio (MSRS), which not only allows programming distributed applications in a pretty straight forward way using the latest development tools, but also lets us simulate robotics applications taking into account physics-based models.
II. MICROSOFT ROBOTICS STUDIO

In December 2006, Microsoft Robotics Studio (MSRS) 1.0 has been released. This development environment for distributed robotic applications is intended to ease the creation, monitoring and usage of robotic programs. Microsoft’s motivation for this step was that today’s robotics industry looks similar to the PC industry in the 1970s. Each company uses another programming language, the robots run on different operating systems and hardware. The reuse of application code between different robot systems is nearly impossible. The concurrency between sensors and actors is difficult to program and requires usually deep knowledge of multithreading programming paradigms. MSRS addresses these issues by means of commonly known .NET programming languages and a library hiding the complexity of concurrency.

The MSRS consists of three major parts: A runtime environment, a 3D simulation environment and a visual programming language (VPL). A collection of about thirty tutorials and samples, and manifest files (profiles) for a variety of the most popular commercial robots, like the LEGO Mindstorms, the KUKA Light-Weight-Robot or the iRobot Roomba rounds off the Robotics Studio.

To support distributed applications in a simple manner Microsoft uses a service-based protocol, the “decentralized software services protocol” (DSSP) [8]. An application consists of one or more services, the basic building blocks in MSRS. These services process the sensory input and orchestrate their results to some actuators. To keep the maintenance as simple as possible, each service can be displayed, configured, started or destroyed in a web browser during the runtime of the application. Of course a service can interact with the user through a self-written GUI or other visual or audio means, too. In addition, a set of standard services exists to facilitate basic tasks, like subscription, debugging or service discovery. Every service possesses an internal state, a global unique identifier, a behavior and a context. For each service, one can additionally define operations to ensure the functionality of the service and to change its state. Several services have to work together to achieve the goal of the application, partnerships to other services can be defined. Thereby, each service knows before his own start only the interface of his partner and its type, but not where the partner service is physically located. The communication between services takes place one-sided or by means of Request / Response messages. Moreover subscription and event notification is supported. With the introduction of the “Concurrency and Coordination Runtime” (CCR) [9], the programming of concurrent applications, as they frequently appear in robotics, becomes easier. Often one must wait for sensory data, while the application should not block. With the CCR it is possible to coordinate messages without the difficult usage of threads, locks and semaphores. By providing interleaves, predicates, joints, and different receivers for messages, some very complex scenarios can be created in a rather simple way.

With the help of a physics-based simulation engine it is possible to create robotic applications, that do not have access to a real robot. Furthermore it is an outstanding possibility to test a program before it possibly damages the sensitive and expensive robot, or even worse, other people. The Ageia PhysX engine, basis for the simulation engine and component of a variety of current computer games, allows photorealistic scenarios with high requirements on physical accuracy. For simulation purpose, one needs to program an entity of an arbitrary robot that is inserted into the simulation engine, and a service controlling the entity. However even the most detailed model can not replace the testing with a real robot. But certain artefacts can already be observed in the simulation before going to reality.

To enable non-programmers or even children to build robotic applications, MSRS is shipped with a visual programming language (VPL). This feature allows to create applications via Drag and Drop without the knowledge of a programming language. The user just needs to know the function of the services he uses and create a sequence of services and activities (e.g. for branches or mathematical calculations) to achieve his goal. The MSRS runtime then creates the corresponding C# code out of the visual representation and executes it.

It appears that MSRS is ideal for educational purposes, but not limited to it. Students can start their first tries with a simulated robot, to get some practical experience besides their theoretical knowledge, without the university to buy a robot or even more for dear money. The creation of non real-time applications was eased a lot, because now one has not to learn a new programming language for each robotic platform, but can use all languages supported by the .NET framework. Still some criticism on the Windows based framework remains: Its high requirements on hardware and the missing real-time support. It’s not too bad, that the simulation engine only runs fluently on a high-end gaming computer, but also in the robot, there’s the need for a fast system, that is not always available. This can be resolved if one has the possibility to create a distributed application where only a small part runs on the robot and the computing time intensive calculations are made on a connected desktop PC. A hardware guide for robotic platforms and a version of MSRS for real-time Windows CE was announced by Microsoft for 2007.

III. SERVICE-BASED ARCHITECTURE

The virtual robot control proposed in this contribution is
The architecture consists of three tiers (see Figure 1 - To emphasize the hierarchical structure of the services and for purpose of clarity, the response messages are not shown in Figure 1). The data tier contains XML files for the configuration of the entities in the simulation environment. With these files, the elements and their controlling services shown in the simulator can be configured. Robots can be composed by an articulated arm, a gripper tool, sensors, and a mobile platform. The service for the simulation then parses the XML description of the scene and automatically generates the robot and its environment via .NET reflection and inserts them into the simulation engine. Furthermore all the robotic and environmental entities can be configured individually (e.g. their appearance or the behavior of their joints). This guarantees an easy configuration of the robot parts without any changes in source code.

In the presentation tier all services for the user interaction with the robot are located. A dashboard can be used to control each single service provided by the framework. Alternatively, interaction devices like a simple mouse or a joystick are supported. Moreover it is possible to write programs in form of services in a high level language like C#. For demonstration purpose, the commonly known “tower of Hanoi” (see Figure 2) algorithm was implemented by means of recursion which is often not available in ordinary robot programming languages. This is a clear simplification for the programming of robots.

The logic tier consists of control services for the real or simulated robot. The main parts are services for the motion planning of Point-to-Point (PTP), linear (LIN) and circular (CIRC) movements. Currently asynchronous, synchronous and fully synchronous PTP and LIN motions are supported. After the calculation of the different joint angles on the path, a response message with the destination angles is sent back to enable a service in the presentation tier to calculate a following movement. All further commands of the same type (PTP or LIN) are just appended to a list of target angles. As soon as the movement stops, a notification is sent to all subscribers to allow other operations to start sequentially. With the help of an inverse kinematics service, which works for the KUKA Light-Weight-Robot (LBR) and other KUKA Industrial robots, the joint angles can be calculated from the Cartesian position and orientation of the Tool-Center-Point (TCP) (inverse kinematics) and vice versa (forward kinematics). Since the KUKA LBR is a seven-axis robot, axis 3 is set as fixed for the sake of simplicity, in order to get an intersection point of the wrist for the last three axes. The transformation is then similar to ordinary 6-axes robots. The up to eight solutions of the inverse problem are sorted by ascending alteration to the current joint angles in order to minimize the changes of the joints. By changing the Denavit-Hartenberg parameters in
the configuration of the articulated arm, it is also possible to use other kinematics. The results of the motion planning are sent to the orchestration service of the robot. It routes the commands to the corresponding controlling service of the robot components in the simulation or reality. Hence the program code can be used for different robots, regardless whether the simulation engine is used or the real robot. The service in the presentation tier again needs not to know the components of the robot, because the orchestration service chooses the right one. Finally the messages reach the controlling services of the robot. For each entity that was inserted in the simulation engine and that needs to change its state, such a service exists providing an interface to ensure the functionality of the robotic part. The mobile platform for example can drive to different positions, rotate around an angle and set some speed or force to its wheels, whereas a simple gripper can just be opened or closed. For the articulated arm, angles can be set for all joints and one can be notified when the arm reaches its final position or runs into an obstacle. Additionally the position and orientation of the TCP can be set or requested. In the case of linear joints, their distances can be set. All these changes call immediately methods of the corresponding entity, which is updated during the next frames in the simulation window.

Besides the above mentioned robots, some more tools, work pieces und environmental entities that can be inserted into the simulation, are in development at the moment. When they are completed it will be possible to replicate whole robot cells. Services to control real robots are the next step. In the future it will be possible to control a robotic platform and to run a C# application on the KUKA robots.

IV. DISCUSSION AND FUTURE WORK

Service oriented Architecture (SOA) does not represent a new approach for software development. Its basic ideas have been around for years. But just recently, probably thanks to the introduction of Web Services, the term ‘SOA’ got its current prominence. With Web Services, the ability to have an application run in a distributed fashion on different hardware platforms showed the full potential of service oriented architectures.

Thinking in Services is generally a good thing during the development phase, since it forces the developer to be clear about roles and interfaces of his software modules. Following service oriented principles leads to code with high cohesion and loose coupling. Just like regular software, now also robotic software build with SOA can use the benefits of high interoperability.
for our controller software.

High software quality means also a lot of testing before the product is deployed. This is another important point, where the MSRS supports its software developers. By providing a simulation environment that includes a physics simulation, much more realistic scenarios can be tested without the need for using expensive robotic hardware.

Currently the simulation of an industrial cell is mainly done to calculate cycle times and to check for collisions. With MSRS it is still possible to calculate cycle times for a cell, and beyond that, it makes the programming of the simulation itself much easier.

In current robotic simulation systems each physical connection between objects (e.g. part in a gripper) has to be ‘programmed’ by some simulation tool specific instruction. This is time consuming and error prone. The MSRS simulation environment eliminates the need for this special programming, since it works by calculating the physical interaction between objects, e.g. a gripper and a part. This does not only make things easier for the simulation programmer, but it is also more realistic. An object that is released on a conveyor for example might be hit unintentionally by the gripper, if the gripper retraction strategy is bad. While this might go undetected with a conventional simulation system, where the gripper might go unseen through the object, in the MSRS simulation system it will instantly show the objects movement caused by the gripper without any special programming. For the application programmer it becomes evident, that he has to take such artefacts into account. Error handling, e.g. for lost objects during gripping, can directly be build in during simulation (see Figure 3).

Collision detection is a popular feature in industrial simulators since it shows dangerous configurations of hardware beforehand. Often it has to be enabled though for certain objects in the scene to analyze the possibility of a collision of those objects. With the MSRS simulation environment, collision detection is not just a feature anymore, but lies at the very heart of its simulation environment. Objects not only collide by default with each other, but they also react physically correct in respect to each other. Since this is supported by new standards for physical simulations, more and more hardware accelerators will support physical simulation and enable the user to have ever more realistic simulation runs of their scenes.

In addition to cycle times and collision detection, MSRS enables the user to do even more for the quality of his product: stress testing. A nice example for a much more realistic testing environment is the loose arrangement of objects in a box. Construct a little slide and let objects slide into a box. Now change for each simulation cycle the starting orientation/position of each object just a little bit - and they will arrange chaotically in the box, just like in real life. By instructing your vision equipped robot to take them out of the box, you can stress test your vision algorithms (MSRS includes a virtual web cam that produces snapshots from the virtual world) by just running the simulation over and over again without real hardware involved.

This shows how the MSRS simulation environment not only facilitates but also extends the simulation possibilities of ones robotic environment. Being already included in the standard MSRS software package, this offers new opportunities for all that are interested in robotic simulations.

Another aspect that comes to mind when looking at these simulation features within the MSRS platform is the possibility of ‘hardware in the loop’. Thanks to the service oriented architecture, you can gradually exchange services that run within the simulation with services on real hardware – for example having a real mobile platform without a real camera ‘seeing’ with a virtual camera and therefore reacting to the virtual world. Mixing simulated hardware with real hardware by only changing few configuration settings opens up many new possibilities for developing and testing different product parts in common simulation scenarios throughout the company.

This combination of a service oriented architecture platform including a physics-based simulation environment makes the MSRS an interesting tool for our future software development efforts.

REFERENCES