Presented at: IEEE ICRA Workshop on Software Development and Integration in Robotics (SDIR VI), pp 6-11, Shanghai, May, 2011

Way of Working for Embedded Control Software Using Model-Driven Development Techniques

Maarten M. Bezemer* and Marcel A. Groothuis** and Jan F. Broenink*

***Abstract*— Embedded targets normally do not have much resources to aid developing and debugging the software. So model-driven development (MDD) is used for designing embedded software with a 'first time right' approach. For such an approach, a good way of working (WoW) is required for embedded software development using MDD techniques.**

This paper discusses the preferred way of working for the development of embedded control software. Control software requires hard real-time support for the loop controllers. These controllers directly control the motor output. Soft and non real-time levels can be used for non loop control related software tasks.

The paper also discusses model structure optimisation techniques, which prevent the complexity of resource scheduling for large robotic setups, These techniques allow the designer to keep his own preferred point of view, as the techniques try to optimise the model into efficient software from a execution point of view. This saves a huge amount of design effort, especially for distributed targets.

I. Introduction

When developing control software for robotic and mechatronic setups, it is convenient to use Model-Driven Development (MDD) [1]. Especially when the target platform is an embedded system which does not have much resources available for all kinds of development aids.

MDD methods result in a reusable model. After the model is used for the required simulations to get the model right and to verify that the controller is able to correctly control the setup, it can be used to derive the actual control software running on the target platform.

Having a model which is validated and working correctly in simulations often results in a 'first time right' implementation of the control software, especially when using the model to generate the actual control software.

It is important to have a complete tool chain from model to execution application when using MDD techniques. A manual step might compromise the validity or correct execution of the control software. This paper shows some new and completed steps in the tool chain presented earlier by Broenink et al. [2].

Section II of this paper gives information about the layered design method used in this paper. Next, the mentioned new steps are discussed in Section III. Starting with the way of working for embedded control software using MDD techniques, followed by a tool chain coverage for the described way of working. After that, the layered design approach is combined with the tool chain and the section ends with model structure optimisation techniques helping the designer with the mapping of the model on the available resources of the target platform. Section IV shows an example usage of the described tool chain. The paper ends with discussions and conclusions.

* Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, 7500 AE Enschede, The Netherlands. e-mail: {m.m.bezemer, j.f.broenink} @utwente.nl

** Controllab Products B.V., Enschede, The Netherlands. e-mail: info@controllab.nl

II. Background

Figure 1 shows a typical architecture used for embedded targets which control robotic or mechatronic setups. It is split into three parts: Embedded software, I/O hardware and the Plant. In this paper the embedded software is discussed.

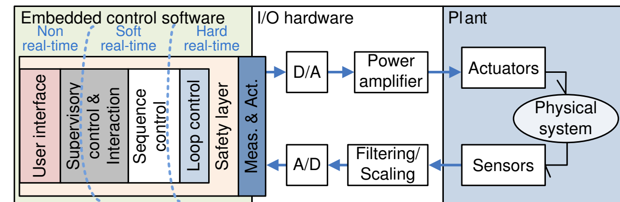

Fig. 1. Software architecture for embedded systems

The embedded software part is split into three layers, each layers defines a real-time property:

- *Hard real-time*; specifies that deadlines must be met, otherwise catastrophic events might occur.
- *Soft real-time*; specifies that deadlines should be met, but if a deadline is not met, the result of this calculation step still holds value.
- *Non real-time*; specifies that software blocks on this layer do not have deadlines to meet.

The software which executes on the non real-time layer gets to use the left-over resources when the software on the other two layers reached their deadlines. This separation of real-time properties is only relevant for control software which needs to run periodically.

The *loop control* block in the figure contain the controllers which control the actuators. These need to be hard real-time in order to make sure that the actuators are fed with new control data each sample period.

The *sequence control* block is an overall controller containing information about the required task. It uses this information to influence the loop controllers in order to have them working together to control the setup and perform the required task.

The *supervisory control* block can be used for long term calculations, which are used by the sequence control to determine what the next task is. Path planning or environment mapping are typical supervisory control tasks.

Using MDD techniques helps to separate the different real-time layers. A generic modeling tool can be used to construct the overall design and connect the different building blocks with each other. Specific tools can be used to design the specific controllers, for example a loop controller can be designed using 20-sim [3]. This is a graphical modeling tool being able to simulate the developed controller using analytical solvers to obtain accurate results.

Model driven development is a good solution when developing embedded software. Usually, embedded targets are low on resources and debugging capabilities, so modeling the software has the advantage of off-target debugging and simulation. When the model is validated, a 'first time right' implementation of the embedded software can be generated from the model. In an ideal situation it really works at the first time, otherwise the amount of on-target debugging efforts to get it working is much lower at least.

III. Method

The first part of this section describes a way of working (WoW) when using MDD tools. The required steps of this WoW, starting at a model and ending with the software application that can be executed on the target, are explained. First from a generic point of view, and then a concrete coverage with tools. Next part discusses the layered approach that is used to obtain the real-time levels. The section ends with a description of model structure optimisation techniques, which should enlighten the workload for the designer for complex or distributed robotic setups.

A. MDD Way of Working

Figure 2 shows the design procedure which is used to develop embedded control software using MDD tools. An earlier version was presented by Broenink et al. [2]. This section elaborates in more detail on a way of working when developing control software for embedded targets, ie the left part of Fig. 1 namely the embedded control software and the peripherals.

The figure consists of 4 steps (numbered 1 to 4) where the tool chain needs to perform some action. Each step is more or less Y-shaped, indicating that two inputs are merged into an updated output. This is repeated for every step until the desired output is obtained.

The top of the figure contains the MDD tools used for developing the controller models and the overall software model. MDD1 is the tool which is used to design the controller models. The tool typically is able to simulate the developed controllers, so it is easy to debug the model before it is executed on the target which does not have elaborate debugging methods.

The MDD2 tool could be used for design of the overall architecture of the software. The model from the previous MDD tool can be included in this tool and the required

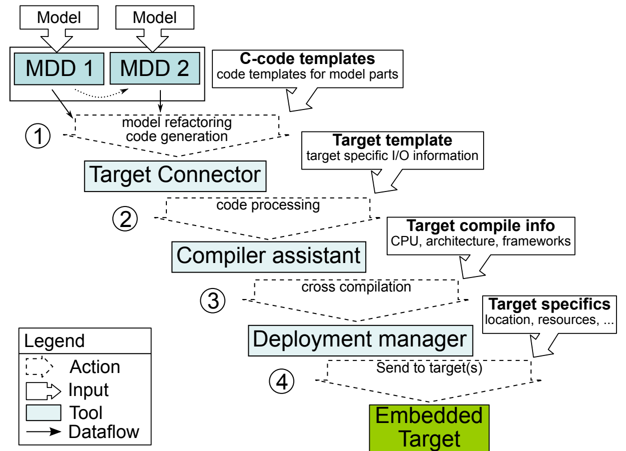

Fig. 2. Way of working when using the MDD tool chain for embedded control software development

controller I/O can be attached in this overall architecture. It should also be able to model the data flows, from the input through the controllers to the outputs, but also internal data flows between different controllers or to the user interface.

In this case there are 2 MDD tools; depending on the actual situation and the different types of controllers even more MDD tools could incorporated in the way of working. Only requirement is, that the tool responsible for the overall architecture is able to make use of the models from the other tools: direct understanding of the model or through code generation.

The first step of the way of working (indicated with a 1 in Fig. 2) incorporates model refactoring and code generation. Model refactoring is optional, but can be used to optimise the models for an execution point of view, more on this in Section III-D. As mentioned, all the MDD tools should be able to generate code from their models. This is achieved using code templates, which contain the code that is required for each model block. By using token replacement such a code block can be adjusted to exactly match the defined properties of the model block. Independent of the used framework(s) the code generation can be kept constant and the templates can be replaced by others when required.

Step 2 involves the extension of the generated model code with target information. External model inputs and outputs are connected to the targets I/O devices. Furthermore, target-specific code required for the execution of the model code on a particular real-time operating system (RTOS) is added.

The result of step 2 is a set of source files that contains all layers from Fig. 1 and target specific information: the embedded control software. Step 3 involves the compilation of these sources with a compiler that produces target-compatible code. Examples of these targets are: (embedded) PC's running an RTOS like QNX or RTAI-Linux, ARM boards and PowerPC cores in FPGAs. This broad range of targets requires the usage of the appropriate cross-compiler to get a compatible executable for each target.

The last step (4) is to deploy the executable to one or more targets. Doing this with a tool in an automated way, might add additional features like logging or debugging depending on the used tool.

B. A tool chain coverage for the way of working

This section shows a tool chain coverage for the described WoW. It is not the one and only coverage, but it acts as an example to shows a possible implementation of the WoW.

At the top of the Fig. 2 two MDD tools are shown: 20-sim will be used for MDD1 and gCSP for MDD2. gCSP is used to develop the overall software architecture and 20-sim is used to design the control software.

As mentioned 20-sim is used to design and simulate the controller models. The 20-sim generated code has the same functionality as the model itself, so it pays off to carefully design and simulate the modelled controller. Using 20-sim results in a 'first time right' most of the times and thus saving valuable debugging time.

gCSP [4] is a graphical CSP-based modeling tool. Communicating Sequential Processes (CSP) [5] is a formalism representing the concurrency between processes. By using this formalism one does not need to think about relations between processes or the execution order of processes. gCSP provides a graphical way of defining these relations and data flows, since using the CSP formalisms directly is too complex for larger target applications. The code generation, in combination with a CSP supporting framework, takes care of converting the model using the CSP formalisms.

LUNA [6] is an example of a newly developed CSP supporting framework, which is used for step 3 of Fig. 2 (and of course the code generation templates are aware of this choice as well). LUNA stands for '*LUNA is an Universal Networking Architecture*'. It is a multi-platform CSP-based execution framework, with support for features like threading and (hard) real-time. The required support for CSP and real-time is explained already. Having threading support enables to fully use multi-core targets and it is also convenient for the model structure optimisation technique described later in this paper. Currently, gCSP is not able to generate code for LUNA yet, so code generation for the CTC++[4] library is used and manually converted to make use of LUNA. The rest of the paper assumes that the code generation is able to generate LUNA code to keep the story apprehendable.

In this example tool chain coverage, steps 2 till 4 are almost completely handled by a tool called 20-sim 4C [2], [7]. It uses a template-based approach to handle specific information for each of the supported targets and software frameworks. A new target or framework can be included by simply adding a new template to 20-sim 4C containing all required target-specific information like driver code.

The code generation step (1) is handled by both modeling tools, only 20-sim supports templated code generation at the moment, but plans are made to update gCSP as well. The generated code uses the LUNA application programming interface (API) to hook into the framework. LUNA provides the CSP based execution engine to execute the (modeled) processes in the right order using its multi-threading support.

Step 2 is completely handled by 20-sim 4C, which uses the target template and the token replacement technique to perform this step.

Step 3 involves the compilation of the embedded control software with a cross-compiler. This cross-compiler is generated by our Embryo build system. Embryo is a fully automated modular build system that can build an RTAI or Xenomai real-time Linux operating system from scratch, including the required cross-compilers for 20-sim 4C. Depending on the requirements of logging, debugging and the availability of target templates for this step, 20-sim 4C or manual cross-compilation is used.

When 20-sim 4C was used for compiling, it automatically sends the executable to the target. Otherwise, this is also a manual step.

C. Layered approach

As discussed earlier and shown in Fig. 1, a layered design approach is required for real-time applications to separate the levels of real-time. These real-time levels can be implemented using a framework that has priority support. The hard real-time code should be executed first to make sure the deadlines are met and thus have the highest priority. After that, the soft real-time code should be executed and when there is left-over time within the period the non real-time code can be (partially) executed. Most operating systems provide thread priority levels, which are used by LUNA to provide the layered approach for the different real-time layers.

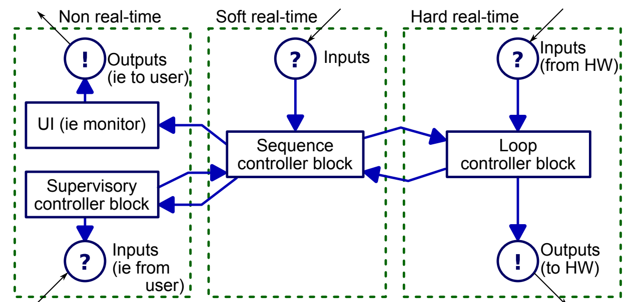

Fig. 3. Example of a layered design

In order to support the discussed layered approach, the MDD tool (gCSP in this case) should be able to support them. The code generation needs to know which process should placed on which layer, especially when there is one big model of the complete application, which is probably desired from a developer point of view. An example of a graphical model using layers, as it could be presented in a MDD tool, is shown in Fig. 3. A legend explaining the used symbols is shown to Fig. 7. All the real-time layers from Fig. 1 are present and contain parts of the software architecture. The controllers are interacting with each other, telling about their status or giving new tasks. Depending on

the application, a control could also have direct I/O with the outside world.

The tool could use properties for each process property to select the layer a process belongs to, or by grouping the processes the layers could be defined. This is shown in Fig. 3, where the layers are shown graphically by using dashed rectangles.

When having this information present, it is fairly easy for the code generation to map the processes of Fig. 3 onto the correct layer and to make sure the real-time properties are effective. This mapping process gets much more complex when the target system has multiple cores or maybe it is even a distributed target. Distributed targets are not too hard to imagine nowadays: a humanoid robotic setup has many joints and it would make sense when each joint has its own embedded controller target. Each of these controllers is supervised by a centralised supervision controller target.

Having a MDD tool which makes it possible to create one model containing all decentralised joint controllers and a supervisory controller, would be very convenient for the developer. The model would contain all elements of the system in one overview, each element is specified by a sub-model. The joint controller could even be made reusable for each joint when it is cleverly designed.

An example of a decentralised setup is shown is Fig. 4. It is a simplified humanoid-like robotic setup with a decentralised controller for each joint. The central controller, in the head, synchronizes all joint controllers and makes sure that each joint controller performs its own piece of the overall task. Comparing it with Fig. 1 shows that the joint controllers are the loop controller, as they need hard real-time guarantees. The centralised controller could be mapped onto the soft or non real-time layers, depending on the functionality of the centralised controller.

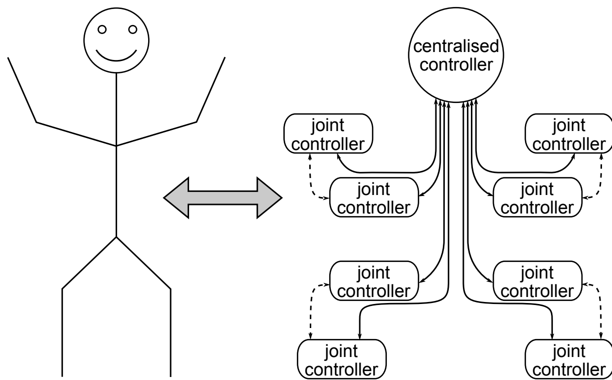

Fig. 4. Example of a decentralised control setup

It would be possible to also have communication between the joint controllers themselves, shown with the dashed lines, in order to perform more complex tasks or obtain an higher accuracy, as indicated with the horizontal lines in Fig. 3.

The code generation now has the complex task to generate code for each decentralised embedded target and to have some form of transparent communication with the centralised supervisory target. The next section discusses model structure optimisation techniques helping with this complex task.

D. Model structure optimisation techniques

By adding a model structure optimisation tool to the tool chain, it is possible to support complex control applications as discussed in the previous section without unnecessary burdening the developer with this complex task. It allows the developer to build models according to his point of view without losing too much efficiency, as the model structure optimisation technique, implemented by the additional tool, converts the model to an efficient software execution point of view, while keeping the available resources in mind so they can be used optimally.

The optimisation technique could determine which processes should be placed on which core, or in the distributed example on which embedded target. An important step in this process is to build a dependency graph of the processes in the model. Processes are dependent when there are data communication flows between them or there is some concurrency relation, like a group of processes should run in parallel. The graph keeps track of these relations and helps grouping the processes into convenient groups which do not have much relations to other groups. Basically, the groups are a kind of separated islands without too much 'external relations' to other islands, as they are expensive compared to 'internal relations'.

To be able to determine efficient groups of processes, model structure optimisation techniques obviously requires extra data, besides the processes dependencies. For example, the amount of data communicated between two processes or the resource usage of each process. Also, the location of certain I/O determines the location of the processes, as some processes need to be able to access the particular hardware providing this I/O. The analysis also needs information about the system, like the available resources of each (distributed) core or the communication costs and available communication bandwidth between each of the cores.

Bezemer et al. [8] presented an automatic model analysis tool, which uses the dependency graph and the information described above to group the processes onto so called 'heaps'. These 'heaps' are groups of closely related processes. Next, these heaps are mapped onto the available cores in the setup, depending on the amount of available cores multiple heaps might get mapped onto the same core.

IV. EXAMPLE

This section shows a simple example using the discussed approach. It is a 2 degrees-of-freedom (DOF) setup, which is basically a platform with panning and tilting capabilities. Using 2 encoders the orientation can be fed to the controllers. The setup can be controlled using a joystick, where one axis influences the pan motion and the other the tilt motion.

A complete model is created using 20-sim, see Fig. 5. The model not only contains the two controllers (implementation shown in Fig. 6), but also simulated inputs, a modeled the software to hardware interface (IO) and a modeled plant

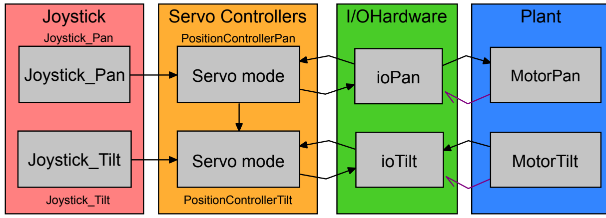

Fig. 5. Top level overview for 2 DOF model

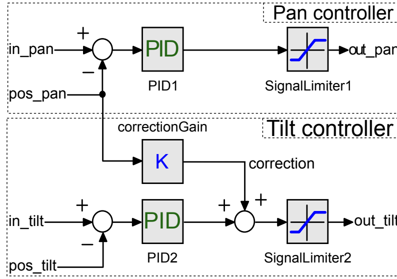

Fig. 6. Pan and tilt controller models

of the 2 DOF robot. Therefore it is possible to completely test and simulate the controllers. After simulations show the model is working correctly, the code generation feature can be used to generate code for the the controller part only.

Next, an overview model is required to connect the two 20-sim controllers to the actual I/O using gCSP, as shown in Fig 7. Each of the top level processes have a sub-model containing the actual implementation. The green box around (groups of) processes show that they are grouped by their concurrent relation. Linkdrivers are used as glue software to connect the gCSP model to actual hardware, in this example they can be seen as I/O drivers. So the *in_pan*, *pos_pan* and *PanProcess* are a group of parallel processes.

The *PanProcess* sub-model, shown in Fig. 8, contains the 20-sim *PanController* model, same goes for the *TiltProcess* sub-model. The *SanityCheck* process checks whether the pan and tilt input values are within a specified range, otherwise they are clipped. This will prevent damage to the actual setup due to high motor velocities. All processes are put on the hard real-time layer as no sequential or supervisory controllers are required.

Using these models, the code generation of 20-sim and gCSP results in a complete application without the need of adding code manually. Before code generation, the model architecture optimisation techniques could have determined an efficient way of mapping the processes.

For example, if a dual core target is used it could decide to put the pan and tilt processes on separated cores and the *SanityCheck* on the core with the most leftover resources. Another mapping possibility is to map the pan and tilt processes on the same core and let the second core run the sanity check and the output handling, so it would be possible to start the sanity check directly after the *PanProcess* is finished and running parallel with the *TiltProcess*.

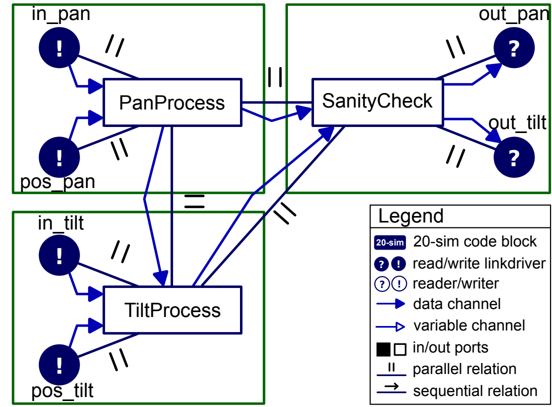

Fig. 7. Top level overview of the 2 DOF setup model

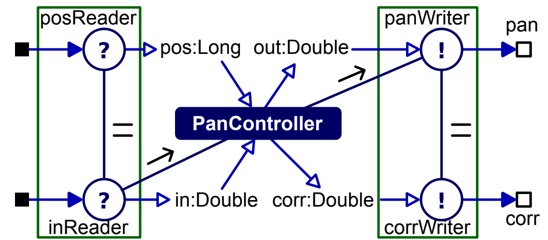

Fig. 8. PanProcess sub-model containing the 20-sim controller

The best solution for an efficient execution implementation of the model depends on the extra information mentioned earlier, consisting of the available target resources, the used resources of each process and communication costs.

V. Discussion and Conclusion

The paper discussed the preferable way of working when developing embedded control software using MDD tools. It can be quite effortless when a good integrated tool chain is available. Flexibility of the WoW is maintained using templates along the route, as new templates can be added to support new target or frameworks.

The example shows that this way of working results in a fairly efficient result. The modeling tools help the developer keeping the overview of the complete application and still be able to simulate the loop controllers, without being forced to think about efficient execution implementation issues.

For example, if one would like to use OROCOS [9] or ROS [10] instead of LUNA as a framework, assuming that those frameworks support al required features, the C-code template need to be updated. All LUNA specific 'hooks' should be replaced with their OROCOS or ROS equivalents. If those frameworks do not support a required feature, it could be added by a custom made library which acts as glue software, or the modeling software should be modified to restrict the supported features. Changing the MDD tool, compared to updating a template, is a hassle and not effortless, so this approach would not be preferable.

The presented way of working supports the layered structure by adding properties to the MDD tools for these layers.

The real-time levels are just an usage example of these layers, other properties could be added to layers as well, as long as these properties are supported by the used framework which is added during compilation.

Model architecture optimisation techniques are really required to support MDD tooling where the designer is able to create a model according to his personal point of view. The optimisation techniques convert the model from the designers point of view to a software execution point of view, without modifying the intended behaviour. We will look into these techniques and are planning to include them into future versions of our MDD tool chain.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 248134.

This research has been partly funded by the Dutch Ministry of Economic Affairs and the province of Overijssel under the "Pieken in de Delta" (PIDON) initiative as part of the TeleFLEX project.

REFERENCES

[1] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, Sept. 2003.

[2] J. Broenink, M. Groothuis, P. Visser, and M. Bezemer, "Model-driven robot-software design using template-based target descriptions," in *ICRA 2010 workshop on Innovative Robot Control Architectures for Demanding (Research) Applications*, D. Kubus, K. Nilsson, and R. Johansson, Eds., IEEE. IEEE, May 2010, pp. 73 – 77.

[3] Controllab Products B.V., "20-sim website," 2011. [Online]. Available: http://www.20sim.com

[4] D. Jovanović, B. Orlic, G. Liet, and J. Broenink, "gCSP: a graphical tool for designing CSP systems," in *Communicating Process Architectures 2004*, ser. Concurrent Systems Engineering Series, I. East, J. Martin, P. Welch, D. Duce, and M. Green, Eds., vol. 62. Amsterdam: IOS press, Sept. 2004, pp. 233–252.

[5] C. Hoare, *Communicating Sequential Processes*. London: Prentice-Hall, 1985.

[6] M. Bezemer, R. Wilterdink, and J. Broenink, "Luna: Hard real-time, multi-threaded, csp-capable execution framework," in *Communicating Process Architectures 2011, Limmerick*, ser. Concurrent System Engineering Series, P. Welch, Ed., vol. 68, no. WoTUG-33. Amsterdam: IOS Press BV, Nov. 2011, [Paper is submitted].

[7] Controllab Products B.V., "20-sim 4C website," 2011. [Online]. Available: http://www.20sim4c.com/

[8] M. Bezemer, M. Groothuis, and J. Broenink, "Analysing gCSP Models Using Runtime and Model Analysis Algorithms," in *Communicating Process Architectures 2009, Eindhoven*, ser. Concurrent System Engineering Series, P. Welch, H. Roebbers, J. Broenink, F. Barnes, C. Ritson, A. Sampson, D. Stiles, and B. Vinter, Eds., vol. 67, no. WoTUG-32. Amsterdam: IOS Press BV, Nov. 2009, pp. 67–88.

[9] "OROCOS website," 2011. [Online]. Available: http://www.orocos.org/

[10] "ROS website," 2011. [Online]. Available: http://www.ros.org/