The Cognitive Interaction Toolkit – Improving Reproducibility of Robotic Systems Experiments

Florian Lier[1], Johannes Wienke[1,2], Arne Nordmann[1,2], Sven Wachsmuth[1], and Sebastian Wrede[1,2]

[1] Cognitive Interaction Technology — Center of Excellence
[2] Research Institute for Cognition and Robotics — CoR-Lab
Bielefeld University, Bielefeld, Germany
http://www.cit-ec.org & http://www.cor-lab.de
{flier,jwienke,anordman,swachsmu,swrede}@techfak.uni-bielefeld.de

Abstract. [PRE PRINT VERSION] Research on robot systems either integrating a large number of capabilities in a single architecture or displaying outstanding performance in a single domain achieved considerable progress over the last years. Results are typically validated through experimental evaluation or demonstrated live, e.g., at robotics competitions. While common robot hardware, simulation and programming platforms yield an improved basis, many of the described experiments still cannot be reproduced easily by interested researchers to confirm the reported findings. We consider this a critical challenge for experimental robotics. Hence, we address this problem with a novel process which facilitates the reproduction of robotics experiments. We identify major obstacles to experiment replication and introduce an integrated approach that allows (i) aggregation and discovery of required research artifacts, (ii) automated software build and deployment, as well as (iii) experiment description, repeatable execution and evaluation. We explain the usage of the introduced process along an exemplary robotics experiment and discuss our approach in the context of current ecosystems for robot programming and simulation.

Keywords: Software Engineering, Experimental Robotics, Development Process, Semantic Web, Continuous Integration, Software Deployment

1 Introduction

Research on autonomous robots and human-robot interaction with systems that integrate a large number of skills in a single architecture achieved considerable progress over the last years. Reported research results are typically validated through experimental evaluation or demonstrated live at robotics competitions such as the DARPA Robotics Competition, RoboCup or RockIn. Given the complexity of these systems, many of the described experiments cannot easily be reproduced by interested researchers to confirm the reported findings [3]. We consider this a critical shortcoming of research in robotics since replicable experiments are considered good experimental practice in many other research disciplines. Despite this observation, robotics has already made significant progress

towards better reproducibility [3]. This trend can mainly be attributed to the following developments. Firstly, diverse "off-the-shelf" robots have become available that ideally only need to be unboxed and powered-on, e.g., the PeopleBot [1], PR2 [6], NAO [9], or iCub [13]. These are often available in simulation. Secondly, there are open source and community-driven software ecosystems, established frameworks and libraries available, such as ROS [14], OPRoS [10] or Orocos [5] which support researchers by providing sophisticated software building blocks. Lastly, dedicated activities towards systematic benchmarking of robotic systems have been carried out in terms of toolkits for benchmarking and publicly available data sets, e.g. the Rawseeds Project [4].

From our point of view, these are promising developments that foster reproducibility in terms of hardware as well as software aspects. However, besides these initiatives, there are also more fundamental methodological issues that prevent reproducibility of robotic system experiments. For instance, Amigoni et al. [2] already point out deficiencies in experimental methodology.This includes the frequently neglected impact on experiments caused by the relationship between individual components and the whole system, as well as the way how publications need to be written in order to improve reproducibility. We identified the four following *issues* that are critical with respect to sustainable reproducibility of robotic system experiments:

i) *Information retrieval and aggregation:* Publications and associated artifacts relevant for reproduction (software components, data sets, documentation and related publications) are often distributed over different locations, like digital libraries or diverse websites. Hence, already the discovery, identification and aggregation of all required artifacts is difficult. Furthermore, this kind of information is typically not available in a machine interpretable representation.
ii) *Semantic relationships:* Often, crucial relationships between artifacts are unknown or underspecified, e.g.: which specific *versions* (`master` or 1.33.7) of software components in combination with which data set, hardware or experiment variant was in use for a particular study?
iii) *Software deployment:* Most current systems are realized using a component-based architecture [15, 10, 7, 17] and usually not all components are written in the same language. Consequently, they do not make use of the same build infrastructure[3], binary deployment mechanism, and execution environment. Therefore, it is an inherently complex and labor-intensive task to build and distribute a system in order to reproduce experimental results. This becomes even more complex when experiments require software artifacts from more than one ecosystem because there is usually no cross-ecosystem integration model.
iv) *Experiment testing, execution and evaluation:* Advanced robotics experiments require significant efforts spent on system development, integration testing, execution, evaluation and preservation of results. This is particular costly if many of these tasks are carried out manually, which is intriguing,

[3] CMake, Catkin, setuptools, ant, maven, etc.

as established methods from software engineering are available to automate these tasks, e.g., based on the continuous integration (CI) paradigm. To the best of our knowledge, so far, these techniques were not widely adopted by the general community for the iterative design, automated execution and ensured repeatability of robotics experiments.

To tackle these issues we introduce an approach for reproducible robotics experimentation based on an integrated software toolchain for system developers and experiment designers. Currently, it supports (robotics) software and simulation environments while physical robots/entities will be introduced in later versions. This toolchain is described in Section 2 and combines state of the art technologies like web-based catalogs and continuous integration methods into a consistent process that facilitates the reproduction process of robotic systems and experiments. After describing the toolchain, we will briefly outline the replication process for a simulation experiment in Section 3 and conclude with a discussion of well-known robotics ecosystems with regard to their support for reproducible experimentation in Section 4.

2 The CITk Toolchain: Concepts and Components

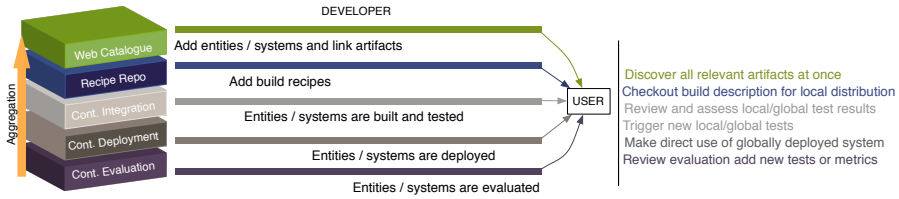

Fig. 1. Cognitive Interaction Toolkit: toolchain and developer workflow

In order to address the aforementioned issues of current robotics research, we have developed an integrated toolchain which accompanies the complete development, reproduction and modification process of robotics systems: the Cognitive Interaction Toolkit (CITk). The CITk consists of a set of software tools which are connected by an underlying software development and reproduction process. This process defines how users interact with our toolchain and has been inspired by current best practices from research and development, especially in robotics. Hence, it ties together existing tools and concepts in an integrated fashion. The general starting point for new users is a web-based catalog. It enables them to browse and search for software components or complete systems and their related artifacts like publications and provides them with the necessary information for the reproduction of these systems. Hence, it addresses the information retrieval and aggregation requirement (i) by means of semantic relationships between the different constituting parts of a complete system and its reproduction process (issue ii). From this catalog, a user who wants to reproduce a system, is directed

towards a build infrastructure which allows to consistently and conveniently reproduce a system based on the CI methodology. Here, we have developed a new, build system independent, solution to easily bootstrap systems based on a CI server by using a generator approach, which reduces the required knowledge and manual work to a minimum (cf. issue iii). Finally, after deploying a system as a software distribution, the web catalog informs the user about tests and experiments that have been performed with the system. A novel state-machine-based testing tool enables users to consistently reproduce these experiments, thereby resolving issue iv. Throughout the whole process, the web catalog forms a central information point for the user. Moreover, tools included in the CITk process are connected with this catalog to either retrieve the required information or push them back in case a system has been modified or even newly created. This fact is visualized in Figure 1. In the following subsections we will describe the fundamental building blocks of the CITk in detail.

2.1 Information Aggregation and Retrieval

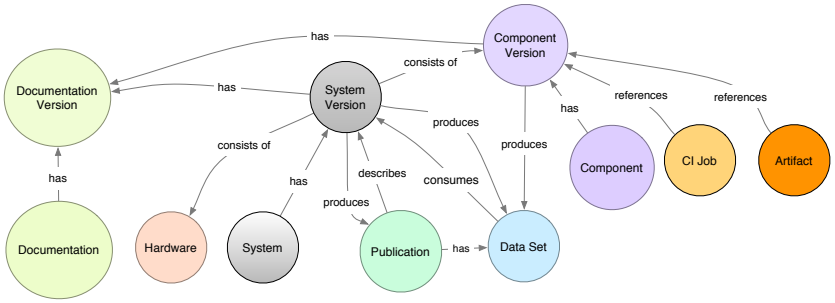

Fig. 2. Cognitive Interaction Toolkit: conceptual data model

In order to realize a web-based catalog which allows convenient information retrieval, we implemented the data model depicted in Figure 2 by extending the Content Management System *Drupal*[4]. Since Drupal already supports the concept of entities, we were able to translate our data model into so-called Drupal *nodes*. A Drupal node provides a container for diverse attributes which are named fields. Fields eventually contain the actual content of a node, such as title, content authors, links to other nodes, files, or text fields (e.g. version). We implemented all required node types according to our data model within the catalog. An exemplary node type for a *component* can be visited here `https://toolkit.cit-ec.uni-bielefeld.de/node/238`. Each component, as well as other node types, assembles basic meta information about the represented entities, such as repository location, wiki pages, component maintainer etc. Moreover, related nodes are linked to the component. For instance a corresponding publication, version number, component releases, systems, or data sets. Finally, nodes and

[4] `https://drupal.org`

their fields are semantically enriched to increase machine interpretability by attaching RDF terms to them, e.g., from DOAP[5] and Dublin Core.

Links between different node types form aggregations of nodes. A prominent example of a node aggregation is the system version type which corresponds to a software distribution for a system to reproduce. Here, a system version node assembles *required components*, interrelated experiments (cf. Section 2.3), manuals, how-tos, and of course data sets and publications. In order to prevent redundant labor with respect to user provided content, most of the catalog's content is imported from the entities' origin locations. Thus, we import required information about a publication by using the Mendeley API and the PUB MODS [18] interface for instance. A user only needs to provide a URL pointing to his/her publication and a corresponding node is created automatically. The same strategy is used to add artifact and build job nodes, here the Jenkins REST-like[6] API is utilized. Besides manual creation/import of entries, content can also be added by using the catalog's REST API and client application[7]. In parallel to the import features, catalog content can be either rendered as HTML including RDFa, pure RDF or JSON to improve automated harvesting and interpretation by search engines or client applications. In a first user-study [11] we demonstrated that our approach of referencing/importing existing sources delivers benefits of re-using data and is perceived as efficient. Furthermore, the web catalog is perceived as useful to help researchers to accomplish their individual goals by providing information in this manner. The overall required effort for importing, respectively adding artifacts, was considered low. The *beta* version of the catalog is publicly available at `https://toolkit.cit-ec.uni-bielefeld.de`.

2.2 Automated Build and Deployment

hile the data model of the web-based catalog already describes the composition of system components, it lacks information on how to technically reproduce a referenced version of a system and its components, e.g. by deploying and executing it. For this purpose, a controlled build process is essential to achieve re-use and reproducability of experimental setups in robotics. Such a build process comprises two distinct aspects: *a*) the build system of individual components, and *b*) the composition of individual component builds into deployable software distributions. For the first aspect, existing ecosystems in robotics often come with custom build systems in order to facilitate the aspect of creating software distributions. For instance, ROS provides Catkin as the build system and NAOqi promotes qiBuild. Both solutions have chosen CMake, a standard build system for cross-platform C++ builds, as their basis. While such a solution is straightforward, it comes with several drawbacks: First, developers have to learn a new technology, which sometimes results in refusal to integrate at all. Second, established build systems, especially for other languages than C++, are locked out.

[5] `https://github.com/edumbill/doap`
[6] `https://wiki.jenkins-ci.org/display/JENKINS/Remote+access+API`
[7] `http://opensource.cit-ec.de/projects/citk`

```
{
  "name":      "openrave",
  "templates": [ "cor-lab", "cmake-cpp" ],
  "variables": {
    "description":     "Open Robotics Automation Virtual Environment...",
    "keywords":        [ "library", "robotics", "simulation" ],
    "repository":      "https://github.com/rdiankov/openrave.git",
    "branches":        [ "master", "latest_stable" ],
    "git.wipe-out-workspace?": false,
  }
}
```

Fig. 3. Recipe for the OpenRAVE [8] toolkit component. Required fields are component name, the project templates (specifying to use a CMake build system in this case), references to the available code branches as well as a URL to the source code repository. Further fields can be used to augment the automatic dependency analysis or customize the build process.

And third, software components developed with such a tailored build system are heavily locked into a specific environment, which prevents the use outside of this environment.

For the aspect of composing software distribution, existing solutions like Catkin, in an abstract view, act as a build script that respects the dependencies of the individual software components by building them in the correct order. While this is sufficient to deploy a software distribution and manually re-trigger the installation of certain components, it does not automatically facilitate the ongoing system development process. For this use case, CI with its emphasis on incremental development and automated testing and reporting is an established technology. However, CI is usually maintained in parallel to the distribution deployment process and as a consequence, build instructions are duplicated between these two distinct processes. Moreover, consistently maintaining a large number of jobs for CI servers that manage a complete software distributions is a complex task that requires a considerable amount of knowledge.

CITk addresses the aforementioned issues by applying a generator-based solution. A newly implemented generator uses minimalistic descriptions of the different software components that belong to a distribution and generates jobs for a CI server, Jenkins in our case. From the descriptions (which augment the data model, cf. Figure 3 for an example) and an automatic repository analysis the dependencies and required build steps are derived. Afterwards, the CI server jobs are generated along user-defined build templates and uploaded to a Jenkins instance. Since templates can be added on the fly, new build systems can be added easily without restricting component developers to certain choices. Moreover, different jobs for either deploying a distribution or supporting the ongoing development can be automatically generated from the same knowledge base, preventing the aforementioned duplication of knowledge. The generated jobs and distributions are optionally synchronized back to the web catalog. Since setting up an appropriate Jenkins server with the required plugins takes some time, we provide pre-packaged installations for new users. As a result, we can use an established technology like the CI server for deploying complete software

systems without knowledge duplication, which results in an improved maintainability. Operating system (OS) dependencies like Debian packages are currently aggregated manually for specific OS distributions. In the future, we will evaluate how to include them in the model to improve consistency and posisiblities for automation.

2.3 Experiment Execution and Evaluation

Besides gathering information about a system and deploying it, a successful reproduction and interaction with the system also includes repeating tests and experiments. This is necessary to ensure the intended responses of complex interactive systems that operate autonomously. It is well-known that sound experiments imply a well-defined experimental protocol. Unfortunately, experiment execution and testing are mostly carried out manually and are thus infrequent and prone to user induced errors. This is especially the case because test setups are usually complicated and require a high level of technical understanding from the operator.

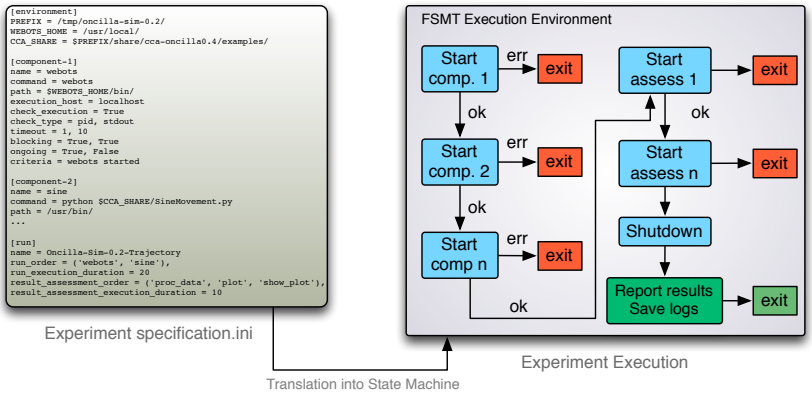

Fig. 4. Simplified conceptual overview of FSMT

Thus, we suggest to convey the concept of an experiment protocol to the orchestration of software components involved in an experiment and to execute, test and evaluate software intensive experiments in an automated manner. For this purpose, we introduce **F**inite **S**tate **M**achine **B**ased **T**esting (FSMT)[8], a software tool that implements the aforementioned suggestions based on finite state machines that defines the experiment execution. FSMT [12] supports automated bootstrapping, evaluation and shutdown of a software system used in an experiment. In order to realize this, it provides definition of environment variables, executable and parameter specification, hierarchical state-based invocation of components, and status (health) checks. A FSMT experiment specification includes three mandatory blocks: environment description, component definition and the run and assessment state (cf. Figure 4). In the *[environment]* block

[8] `http://opensource.cit-ec.de/projects/fsmt`

an experimenter may assign experiment-specific values to variables, these are appended to the existing set of environment variables. Components are specified in *[component-*]* blocks. Here, paths, executables (scripts), and status check conditions are defined, e.g., whether the PID of a specific process is present within a given time frame. Furthermore, FSMT may check the stdout and stderr of components for a given prompt, again, within a specified time frame. Based on the result of multiple status checks, FSMT will block further execution of the state machine or, in case a criterion is not satisfied, stop the experiment to prevent subsequent failures. In the *[run]* state, the actual experiment is conducted, which means that all required components of a system are running (verified) and data is recorded. The recorded data may consist of system specific containers, such as a rosbag[9] or component logs that are recorded for all components by FSMT. In the assessment state, the recorded data is evaluated by assessment components. Here, experimenters may provide scripts or binaries to evaluate data gathered in each run or trial, e.g., plot specified data points. The presented formalization of an experiment protocol allows to consistently reproduce test results in an automated fashion. This makes it a perfect candidate for CI and enables inexperienced experimenters to run tests and experiments consistently. Currently, FSMT supports sequential as well as parallel execution/evaluation of components. However, since it is based on a generic state-machine based execution environment (SCXML), additional conditions like *retrying or looping* are planned as well as distributed execution.

3 Use-Case: System and Experiment Reproduction

After describing the overall structure and components of the CITk approach, we will now outline a typical use case in order to demonstrate the steps required to reproduce a system that has been modeled in the catalog. The reader is encouraged to follow these steps on his/her own Linux computer. In our example, a user wants to reproduce experiments with the Oncilla quadruped robot [16] in a simulation environment. The first step for the user is to browse our online catalog and to load the page describing the system, which is `https://toolkit.cit-ec.uni-bielefeld.de/systems/oncilla-quadruped-simulation`. This page contains a general description of the system as well as a set of links to specific versions of the system. The user will continue to the version he/she wants to reproduce (0.2 in our case) to get details about the included software components as well as required dependencies for the Linux computer. Hence, the first step is to install the required dependencies using the package management tool of the Linux distribution[10]. Afterwards, he/she follows the link to the build generator recipes comprising the system and downloads the required recipes. In order to generate CI server jobs from the recipes a Jenkins installation as well as our generator are required. Download instructions for a pre-packaged environment

[9] `http://wiki.ros.org/rosbag`

[10] We use as many standard software components from recent Linux distributions as possible.

are included on the website and the user follows these instructions to install the environment on his/her computer. After starting the local Jenkins instance, the generator needs to be invoked to configure the CI server with the jobs for the distribution. The last step is to start the installation process and to open the web page of the local Jenkins instance and to start the orchestration job for the distribution. After this job has finished, the system is reproduced on the user's computer.

Now, the user can focus on reproducing the experiments that have been defined for and deployed with the system. These experiments are available as configuration files for the FSMT testing framework. FSMT and the configurations have just been installed as parts of the software distribution for the system. The web catalog for the system version lists the available experiments and each experiment comes with a description of how to execute it. In our case this is merely a command line to launch FSMT. The user executes the described commands and FSMT reports the results through a report (xunit for instance) and the process return code. After the experiment has been executed successfully, the resulting output, e.g., in form of a plot, can be found on the user's computer (as well as all required logs and data files). Besides the FSMT report, the generated plot can be compared to a reference plot from the catalog entry of the experiment. If everything worked correctly, the system is reproduced on the user's computer and he/she can start to modify it according to his/her needs or define new experiments based on the existing system.

4 Related Work and Discussion

In order to relate our approach to exiting work, we will quantitatively examine the prominent ROS and iCub ecosystems with respect to support for system *reproducibility* based on the four issues we have identified in the introduction: i) information retrieval and aggregation, ii) semantic relationships, iii) software deployment, and iv) experiment testing, execution and evaluation. In the **iCub** ecosystem, a wiki[11] is the primary source of information. The wiki *lists* different kinds of information, such as links to user manuals, source code repositories, papers, related (software) projects and summer schools. The main types of artifacts are iCub modules, applications and tutorials. The corresponding documentation is automatically parsed from source code (Doxygen). However, there are no explicit descriptions of existing "demo" systems (versions respectively) as a whole and their included components. Subsequently, system artifact aggregation and interlinking is not provided. Nevertheless, existing iCub applications and modules are assembled and well documented. The iCub ecosystem can be either built from source or by using pre-built binaries for Debian/Ubuntu Linux distributions and Windows. This means that systems are implicitly modeled by package dependency resolution. Furthermore, a central CI server[12] provides an overview of

[11] `http://eris.liralab.it/wiki/Main_Page`
[12] `http://dashboard.icub.org/`

the current state of iCub related software. The build system for the iCub ecosystem is based on CMake. With respect to system experiments, the wiki features an example of checking whether a system installation has been successful or not, which can be valued as a basic system experiment. In this example, multiple iCub components are started manually and a provided data set can be replayed. However, based on the example documentation it remains unclear how to assess the outcome of the system installation check. Additionally, there is no support for automated execution or evaluation.

In the **ROS** ecosystem, a wiki[13] is also the main source of information. It contains structured information about installation, tutorials, distributions, robots, packages, libraries, papers, books, events and more. While there are no explicit system descriptions, ROS features system distributions including a set of versioned ROS stacks. Therefore, systems are also implicitly modeled as stacks or meta packages. Fortunately, ROS provides extra wiki sites for a few publications that aggregate source code, data sets and usage examples (for a specific distribution) to reproduce the results published in the attached papers. Furthermore, there are wiki pages per distribution, stack and (meta) package, i.e., for `pr2_common`[14]. These pages contain basic information, e.g., maintainer, license, website, source code location and links to related/included packages. Basic interlinks between artifacts, i.e., stacks, related packages, dependencies, documentation and source code locations are supported (cf. `http://wiki.ros.org/turtlesim`) — semantic linking is currently not supported. In recent versions, ROS introduced its own new build system called Catkin that is based on CMake. By using Catkin, developers can easily setup and deploy their own ROS components and even add them to a ROS distribution. ROS also provides a CI build farm to which developer packages can be added via ROS-bloom[15], a release automation tool. In order to start ROS-based systems automatically, ROS features roslaunch, a tool for launching multiple ROS nodes locally and remotely. In a roslaunch file, rostests can be integrated. Therefore, ROS features a mechanism to automatically start and test a stack or package. Explicit experiment descriptions are currently not present in the ROS ecosystem.

In general it appears to be a good practice to publish information about robotics software and the corresponding ecosystem on a structured website. Unfortunately, the assembled information is often not complete with respect to system reproducibility due to a lack of explicit system descriptions, aggregation of all necessary artifacts and the specification of an experimental procedure in both examined ecosystems. Since ROS already features websites which assemble at least source code and data sets for a specific publication, we are confident that this way of information provision is beneficial. However, semantic linking and thus machine interpretability of artifacts is broadly neglected in the examined ecosystems. Systems and system versions are modeled implicitly, but are not visible/marked as such on the websites. Moreover, systems are not associated

[13] `http://wiki.ros.org/`
[14] `http://wiki.ros.org/pr2_common`
[15] `http://wiki.ros.org/bloom`

with an experiment protocol or course of action. Not surprisingly, CI plays an important role in both ecosystems but is not considered for local usage, e.g., for decentralized development, testing and distribution. In case of ROS this means that a developer must comply to the ROS release cycle time and server capacity. However, the sponsorship of an automated build infrastructure and tools to automatically create build jobs (cf. ROS-bloom and Section 2.2) reduces the amount of expert knowledge and is thus also considered beneficial. In contrast to our approach, ROS and iCub distributions can be installed via source builds (*not recommended* as stated in the ROS wiki) but also via binary distributions that simplify and speed up installation time. On the other hand, binary packages often raise typical issues such as requiring root permissions for installation, the install prefix is fixed and creating binary packages for diverse operating systems and flavors is a huge effort. With respect to build systems both ecosystems are based on CMake, which facilitates cross-platform compatibility, but also, in contrast to CITk, restricts the number of integrable third-party build tools. This is especially crucial because robotic systems/experiments often incorporate artifacts from more than one ecosystem. Finally, experiment specification, orchestration, automated execution and evaluation is not supported by either ROS or the iCub infrastructure.

5 Conclusion

We introduced an approach for reproducible robotics experimentation based on an integrated software toolchain for system developers and experimenters. It combines state-of-the-art technologies into a consistent process that facilitates the reproduction of robotic systems and experiments. We briefly outlined the replication process for a simulation experiment and discussed the benefits of the approach in comparison to well-known robotics ecosystems and their support for reproducible experimentation. Future work will focus on providing the complete toolchain as open source to the community, extending the build generation with classical continuous integration and deployment features for local development and extension towards modeling hardware components and versions as part of a system description.

Acknowledgements: This research and development project is funded as part of the Center of Excellence Cognitive Interaction Technology (CITEC) at Bielefeld University and by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster Competition "it's OWL" (intelligent technical systems OstWestfalenLippe) and managed by the Project Management Agency Karlsruhe (PTKA).

References

1. PeopleBot datasheet. `http://www.mobilerobots.com/Libraries/Downloads/PeopleBot-PPLB-RevA.sflb.ashx`. visited: 2014-05-19.

2. F. Amigoni, M. Reggiani, and V. Schiaffonati. An insightful comparison between experiments in mobile robotics and in science. *Autonomous Robots*, 27(4):313–325, 2009.
3. F. Amigoni, V. Schiaffonati, and M. Verdicchio. Good experimental methodologies for autonomous robotics: From theory to practice. In F. Amigoni and V. Schiaffonati, editors, *Methods and Experimental Techniques in Computer Engineering*, Springer Briefs in Applied Sciences and Technology, pages 37–53. Springer International Publishing, 2014.
4. A. Bonarini et al. RAWSEEDS: Robotics advancement through web-publishing of sensorial and elaborated extensive data sets. In *IROS'06 Workshop on Benchmarks in Robotics Research*, volume 6, 2006.
5. H. Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.
6. S. Cousins. ROS on the PR2 [ROS Topics]. *Robotics Automation Magazine, IEEE*, 17(3):23–25, 2010.
7. S. Cousins, B. Gerkey, and K. Conley. Sharing software with ros [ROS Topics]. *Robotics & Automation Magazine*, 17(2):12–14, 2010.
8. R. Diankov. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.
9. D. Gouaillier et al. Mechatronic design of NAO humanoid. In *Proc. Int. Conf. on Robotics and Automation*, pages 769–774, 2009.
10. C. Jang et al. OPRoS: A new component-based robot software platform. *ETRI journal*, 32(5):646–656, 2010.
11. F. Lier et al. Facilitating research cooperation through linking and sharing of heterogenous research artifacts. Proc. 8th Int. Conf. on Semantic Systems, pages 157–164. ACM, 2012.
12. F. Lier, I. Lütkebohle, and S. Wachsmuth. Towards automated execution and evaluation of simulated prototype HRI experiments. Proc. 2014 ACM/IEEE Int. Conf. on Human-robot interaction, pages 230–231. ACM, 2014.
13. G. Metta et al. The iCub humanoid robot: An open platform for research in embodied cognition. In *Proc. 8th Workshop on Performance Metrics for Intelligent Systems*, pages 50–56, New York, NY, USA, 2008. ACM.
14. M. Quigley et al. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.
15. P. Soetens. *A software framework for real-time and distributed robot and machine control*. PhD thesis, Katholieke Universiteit Leuven, Faculteit Ingenieurswetenschappen, Departement Werktuigkunde, 2006.
16. A. Sproewitz et al. Oncilla robot, a light-weight bio-inspired quadruped robot for fast locomotion in rough terrain. In *Symposium on Adaptive Motion of Animals and Machines*, pages 63–64, 2011.
17. J. Wienke and S. Wrede. A middleware for collaborative research in experimental robotics. In *2011 IEEE/SICE Int. Symposium on System Integration*, Kyoto, Japan, 2011. IEEE, IEEE.
18. C. Wiljes, N. Jahn, F. Lier, T. Paul-Stueve, J. Vompras, C. Pietsch, and P. Cimiano. Towards linked research data: An institutional approach. Number 994 in 3rd Workshop on Semantic Publishing, pages 27–38, 2013.