Humanoid Control Module: An Abstraction Layer for Humanoid Robots

Marc Bestmann
Department of Informatics
University of Hamburg
Hamburg, Germany
bestmann@informatik.uni-hamburg.de

Jianwei Zhang
Department of Informatics
University of Hamburg
Hamburg, Germany
zhang@informatik.uni-hamburg.de

***Abstract*—Humanoid robots promise a better integration of robots into our everyday life, but they come with additional challenges when compared with other mobile robots. We present a novel approach to simplify their usage by handling these challenges with the *Humanoid Control Module* and show its utility in the RoboCup Soccer Humanoid League. The paper also discusses previous approaches to this problem from inside and outside of the RoboCup domain.**

***Index Terms*—Humanoid, Robot, Architecture, Falling**

I. Introduction

The field of mobile robots is divided into multiple subclasses. One of the most commonly used types of mobile robots are wheeled robots. They are comparably easy to control since they can move on flat terrain without falling and their drive system hardware is simple. Naturally, research in navigation and path planning is mostly conducted on them. Walking robots, on the other hand, especially bipedal ones, are more difficult to control. They can fall quickly, even on flat surfaces, and their drive system hardware is more complex, consisting of kinematic chains with multiple degrees of freedom. Typically, it is necessary to handle these low-level error sources during performing high-level tasks explicitly, thus preventing the direct use of standard implementations, which were initially created for wheeled robots, e. g. path planning. Furthermore, teleoperation does not work on humanoids as simple as on wheeled robots. Some parts, especially fall management, have to be done autonomously on the robot, as no user can act reliably quick enough.

This work presents an approach to enable teleoperation as well as the usage of standard navigation algorithms, originally developed for wheeled robots, on legged robots. The *Humanoid Control Module (HCM)* handles the specific problems of a bipedal robot and thereby introduces an abstraction layer on which high-level algorithms can work as if it was a wheeled robot. It was initially developed for the RoboCup Soccer Humanoid League [11] and has shown its utility there, however, it is applicable to any humanoid robot performing complex tasks.

This research was funded partially by the German Research Foundation(DFG) and the National Science Foundation of China (NSFC) in project Crossmodal Learning, DFG TRR-169/NSFC 61621136008.

We identified six tasks that have to be handled to provide a useful abstraction from the low-level problems. Two of these tasks also exist on wheeled robots, although the implications are different on walking robots. First, hardware problems have to be dealt with. This includes loosing connection to servos and sensors and sudden hardware failure. Contrary to a wheeled robot, this is critical on a walking robot since loosing control of any leg-joint will usually lead to a fall and thus possible damage. A similar difference can be seen with manual stops. They are an essential feature for the safety of the robot, its environment and possibly humans. While on most robots, the motors are simply stopped, or the power is cut, this is not a feasible option on bipedal robots. Since it could, again, lead to a fall and thus possibly even more significant damage. Therefore, more sophisticated methods have to be applied.

Ultimately, falls are not preventable in every situation, making it is generally necessary to manage them. Preferably by initiating counter measurements, e. g. capture steps, or if a fall is not preventable, by safety measurements, e. g. turning the head into a safe position before hitting the ground. After a fall, the robot has to get up again, to be able to perform any high-level tasks. Therefore, a stand-up motion has to be performed. Furthermore, there are multiple software components, which may try to control the same joints at the same time, e. g. when the high-level task wants to control the head to look around, but the falling detection wants to put the head into a safe position at the same time. This can result in unwanted motions and even safety issues. Therefore, a *mutex* has to be put into place that controls which goals are forwarded towards the hardware. Additionally, it might be necessary to inform the high-level behavior of the current state of the HCM to improve its performance, e. g. a self-localization might want to adjust its belief after a fall. Therefore, a semantic state of the robot has to be provided. To sum up, there are two tasks (hardware error handling and manual stop) that have to be managed differently from wheeled robots and four tasks (falling, standing up, joint goal mutex, and semantic robot state), which are entirely new.

In this paper, we present an approach to solve these tasks. Starting with related work in Section II, followed by our approach to tackle the problem in Section III. An exemplary implementation is then described in Section IV, followed by

an evaluation in Section V. The paper then concludes with Section VI.

II. Related Work

The HCM offers a hierarchical decomposition of the system, which is generally regarded as desirable [23, p. 284] and which is widely used in software architecture for robots. One frequently used approach for autonomous robots is the *three-tier (3T)* architecture [16]. It divides the system into three layers, each with its own control loop frequency. The highest layer (deliberative tier) creates a plan to achieve overall goals. Based on this, the second layer (sequencing tier) invokes skills, e. g. closing a gripper, provided by the third layer (reactive tier). In Section III we will show how our approach relates to this general architecture.

Some work has been done to create architectures specially designed for humanoid robots. The RoboCup Soccer team NUbots developed a system called *motion manager* in their *NUPlatform* architecture that decides which motion should be performed based on the current sensor data and a queue of jobs that should be performed [12]. Normally the jobs are executed until their end and then the next one is performed, but based on sensor data the motion manager may invoke a fall protection and stand up motion, which interrupts current jobs. Different movement handlers can also lock joint groups. The motion manager provides methods to get information about joint groups that are currently in use but does not provide a semantic state of the robot. While this system tried to introduce a mutex mechanism for the joints, it did not work [9]. Each movement handler had to check by itself if the joint group was free or not. If the movement handler did not perform this check, it can lead to falls. Furthermore, the indirect accessing of the movement handlers leads to access paths that are difficult to maintain. The complete system was later replaced by *NUClear* [8] in which the motion manager was replaced by an extended form of subsumption logic [2] to allow arbitrary access between skill modules and the hardware. In this system, no explicit mutexes are used, but skills with higher levels of importance, e. g. falling prevention, subsume other joint goals.

Another approach to implementing the motion software part in RoboCup Soccer is to create a single closely coupled module which is controlled by the higher levels [5] [4]. This simplifies the mutex problem but leads to a more monolithic approach, where single parts of the motion, e. g. the walk engine, can not be as easily changed.

Some teams are running the software for controlling the robot's motions on a separate microcontroller [21]. This can decrease latencies but does not provide as much computing power and makes development more difficult.

A large part of research on teleoperating humanoid robots was done during the DARPA Robotics Challenge (DRC). The team THOR implemented multiple Finite State Machines (FSMs) which control different kinematic chains of the robot, such as head, arms, and body [24]. Finally, a *Motion FSM* adds a torso movement compensation for stability and an emergency stabilization. Since falling detection and avoidance is important [10], different approaches were investigated. If the disturbance is small, correcting a few joints can be enough, e. g. using ankle and hip strategy [15]. Stopping the robot's gait can also resolve perturbations to some degree [20]. Larger perturbations can be handled by doing capture steps, where the robot places its next step into a capture region that leads to stopping the robot [18] [14]. In any of these methods, a fall may not be prevented if the disturbance is too large. This case has to be detected and the robot has to be controlled to minimize the damage. For damage minimization, *Ukemi motions* originating from martial arts have been proposed [7]. They try to lower the center of gravity, spread the force of impact over a larger area or time frame, and try to land on specific parts of the robot to reduce the damage.

III. Approach

First, the overall software architecture of the approach is presented in Section III-A and afterwards details about the HCM are discussed in Section III-B.

A. Overall Architecture

The tasks described in Section I depend on a high control loop frequency, as reaction time is crucial, especially in falling scenarios. Making such a decision on a deliberative or sequencing tier does therefore not fit. It would also contradict the goal of reusing standard software components for wheeled robots belonging to these tiers. Therefore, the HCM has to be on a lower tier. Furthermore, we wanted to ensure a real mutex for the joint control, to prevent problems similar to the ones of NUPlatform. One approach would be to use a subsumption strategy, similar to NUClear, but this would not provide a clear semantic state of the robot since it is not a stateful approach. Therefore, we propose a four-level architecture that places the HCM between skills and the hardware driver (see Figure 1). The deliberative and sequencing layers remain explicitly untouched to keep them replaceable with any software designed for wheeled robots. The skill layer remains structurally the same, but when porting software from a wheeled robot onto a bipedal one, some skills have to be adapted, especially replacing the wheel control by a walking engine. The HCM takes in the joint goals of all skills and applies a mutex on them or replaces them with other goals, in case of falling. Any type of joint goals, e. g. positions or forces, can be used. Below the HCM, there is the typical hardware abstraction, which remains untouched, as there is no interface change to this.

The data flow is based on message passing. This gives a clear understanding of which component is receiving what data, which is important for the mutex. Furthermore, it allows an asymmetric interface between skill and hardware abstraction layer. This way, sensor data can directly be passed from the hardware abstraction layer to the skills, but joint goals have to go through the HCM.

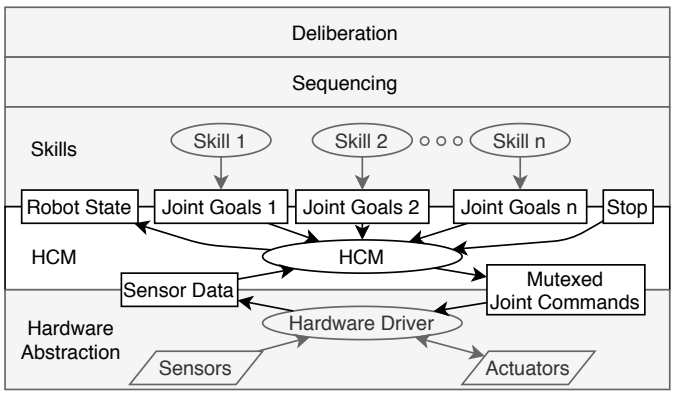

Fig. 1. Position of the HCM in a classical three-tier approach. It is located below the skill layer, to enable mutexing of joint goals and quick reaction for fall handling. The robot state and the manual stop are added as an interface to communicate with high-level behavior. The other layers (**gray**) remain untouched and thereby allow an easy migration of existing software parts.

B. HCM Architecture

The HCM itself has to decide in which semantic state the robot currently is (e. g. walking) by using sensor information (e. g. IMU), servo feedback (e. g. joint torque) and the joint goals from the different high-level tasks. Based on this state, the HCM either forwards all applicable goals from the skills, or replaces them with other goals. The state is also published as information for other high-level components.

As this is a typical decision-making problem, multiple architectures are available. We decided on using the Dynamic Stack Decider (DSD) [17] which is a light-weight framework for decision making that is clearly hierarchic and stateful. It works similar to a decision tree with additionally remembering its own state through a stack of previous decisions.

In this case, there are multiple decisions checking the current state of the robot in a hierarchical way (e. g. is it falling, is there a hardware problem) and actions (e. g. do fall safety measures, stand up). This state is influenced by the sensor information, the high-level task commands, and the user safety stop.

Based on the state of the DSD, a semantic state of the robot (e. g. falling) is provided which can be used by the higher level software parts. Furthermore, the HCM decides on which joint goals can be forwarded to the hardware, based on the current state of the robot (e. g. no control while being in safety stop), and if multiple high-level tasks are compatible (e. g. the head can be controlled at the same time as the legs by different skills). The resulting joint goals are then forwarded to the hardware driver.

IV. Implementation

The intended main usage of the HCM is in the RoboCup Soccer domain [11]. Since it is a multi-agent scenario where complex high-level behavior, including tactics, is necessary, having an abstraction from the low-level handling of the robot leads to a clearer and more understandable software structure. Furthermore, the soccer field is flat, making typical approaches for wheeled robots applicable.

A. Robotic Platform

The approach was implemented on the low-cost *Wolfgang-OP* robot platform with 20 degrees of freedom [1]. It is a successor of the Nimbro-OP [22] and similar in its mechanical structure. An Intel NUC is responsible for computation of all motion-related tasks, such as walking, hardware communication and also the HCM.

Two modifications were made to prevent damages to the hardware in cases of falling. A 3D printed elastic element was inserted between the robot's shoulders and arms (similar to [13]). This prevents gear damage in cases of lateral forces on the arm, as the arm will rather bend and make contact with the torso than apply torque on the servo. Elastic bumpers were installed on the front and back of the torso to lessen the impact forces that act in cases of falls in the sagittal direction. Thereby, the robot can fall without damage in any direction if it moves its arm and head joints to the correct position.

B. Overall Implementation

The complete system was implemented as a publisher/subscriber architecture using ROS [19], as it is open source and the widest spread middleware in robotics. ROS allows easy modularization of software into nodes that communicate with each other through asynchronous topics. Due to its widespread use, implementations for many standard approaches are available.

In our implementation, there is one main high-level behavior, called *body behavior*, which decides where the robot should go, what it looks for, and when it should perform actions, e. g. a kick. It works on a high abstraction level, as all lower-level tasks are handled by the other layers. The navigation goal is handled by the standard ROS package for navigation called *move base*. It was originally intended for navigating wheeled robots, but the HCM makes it is possible to apply it to our bipedal robot. Move base plans a path to the provided goal, including avoiding obstacles that are detected by the vision system of the robot. It is possible to choose from multiple different planners or to implement a new one. We use the *DWALocalPlanner*, which is based on the dynamic window approach [6]. Typically, a footstep planning approach is used in bipedal path planning. This would also be possible by implementing a local planner, but was not necessary in our use case since the environment is completely flat. Furthermore, we want to evaluate how well the classic wheeled approach performs on a bipedal robot. Move base publishes velocity commands, represented by a speed in x and y direction as well as a turning speed. The walk engine publishes corresponding joint commands to reach this velocities. This engine uses the robot state published by the HCM, e. g. to reset its walk phase after a fall. The body behavior may also invoke full-body actions, such as kicks or throw-ins, directly. Kicks are performed by an engine which takes sensor feedback into account to stabilize the robot during

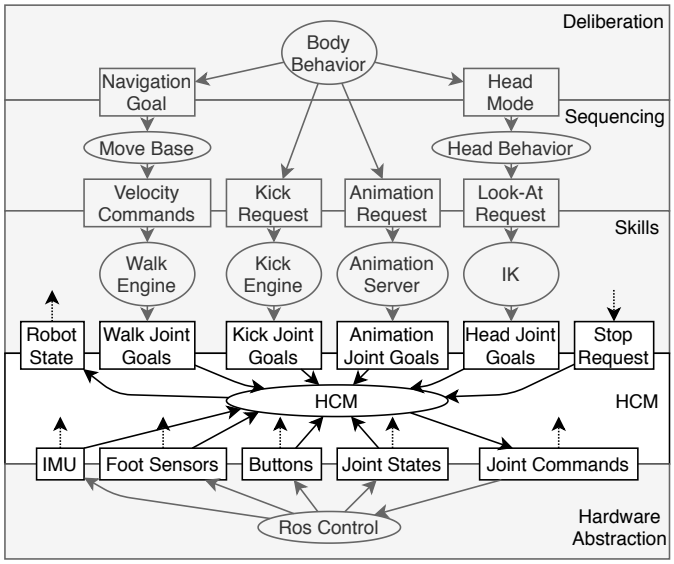

Fig. 2. Overview of the implementation with ROS nodes (**ellipses**) and topics (**rectangles**). The HCM is added between the skill and hardware layer (as described in Figure 1), thus no further adaption of the other layers is needed. The dotted arrows represent direct connection to any other node. The Sequencing and Deliberation Layer are replaced in case of teleoperation by a simple interface for the human.

kick motion. Other actions are performed by an animation server, which interpolates a predefined keyframe animation and publishes joint goals until it is finished. A third source for joint goals is the *head behavior*. It moves the head to gather information about the soccer field. It can be set to different modes by the body behavior, e. g. track the ball, search for goals or search for field features to localize yourself.

The HCM also receives information of the current state of the robot from the *ros control* [3] node, which is the standard ROS hardware interface. It provides data from the IMU, the foot pressure sensors, and the joint states. Furthermore, it accepts the HCM's goals for the resulting joint commands and controls the hardware accordingly. The ros control node offers the same interface for the simulator and the actual robot, thus making it easy to test the software on both. An overview of the architecture can be seen in Figure 2.

C. HCM Implementation

The HCM node itself consists of two main parts: the DSD which is updated in a loop at the same frequency as the hardware interface is running and the call back functions which are invoked asynchronously when joint goals are received from the high-level behaviors. If the received goals can be executed on the robot, given the current state of the DSD, they are directly forwarded to ros control inside the callback, to minimize the delay introduced by the HCM. Otherwise, they are dropped. The HCM provides the following states: *Animation_Running, Controllable, Falling, Fallen, Getting_Up, Kicking, Picked_Up, Record, Shutdown, Startup, Stopped, Walking* .

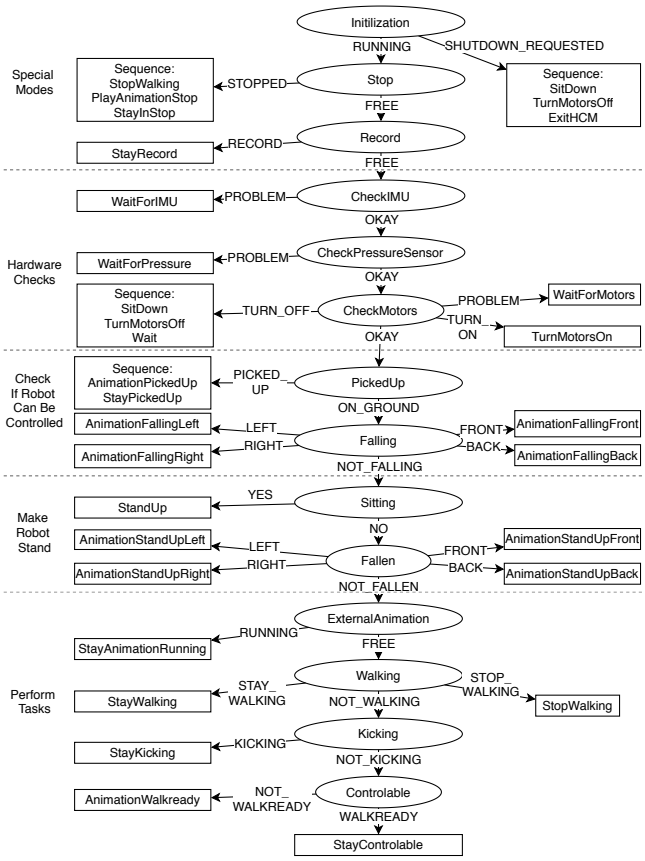

Fig. 3. Structure of the HCM's DSD with additional grouping of parts for clarity. Ellipses represent decisions and rectangles actions.

The DSD consists of 14 decisions which are ordered strictly hierarchically and are reevaluated at every iteration from top to bottom (see Figure 3). These decisions can be grouped into five objectives.

First, special modes or requests are checked. This can be a shutdown request, a stop request or the *record mode* that is used to create keyframe animations and disables the HCM. During runtime, the current state of the HCM can be visualized graphically.

The second group consists of the hardware checks. The passed time since the last sensor reading is used to decide if the IMU and the foot pressure sensors are working. The same mechanism is also used for the servo motors, but additionally, it is checked if the power should be turned on or off. This is useful to power the robot's motors down automatically after being not used for some time, thus preventing unnecessary wear if the user forgets to do so. When the robot receives new commands, the motors are activated again automatically.

The third group checks if the robot can be actuated. If the robot is picked up, it should not move to allow easier handling by the human. This check needs to be done before fall checking since sudden movements during carrying may trigger the fall detection.

After completing these decisions, it is clear that the robot can move, but it is still necessary to check if it is standing.

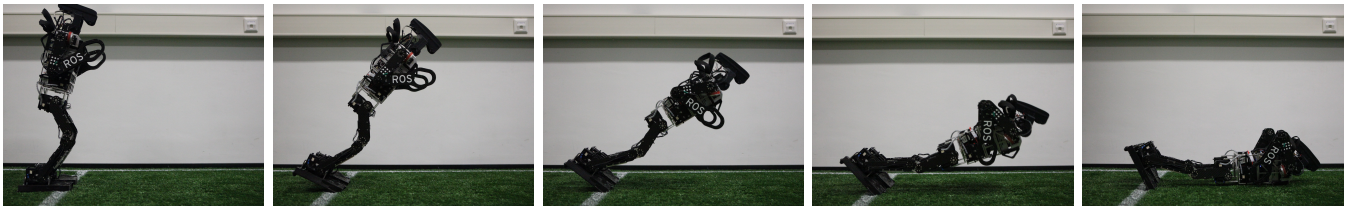

Fig. 4. Image sequence of a frontal fall with HCM active. Between pictures two and three, the robot recognizes the fall and moves arms and head into a safe position. It reaches this position in picture four, before hitting the ground.

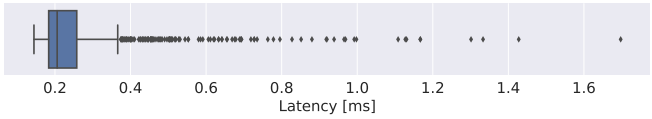

Fig. 5. Latency of joint goals derived from the additional ROS message transfer that is needed to pass the goals through the HCM. The mean is low with 0.25 ms but single outliers go up to 1.7 ms.

Therefore, the fourth group performs different stand-up motions if necessary.

The last group is finally performing the actual tasks of the robot. Keyframe animations can be played, the robot can walk and kick. As a last decision, it is checked if the robot is in the *walkready* pose. This is the standard idle pose of the robot, from which it can start to walk or to perform other tasks.

D. Fall Handling

The fall handling is implemented in a simple but robust way. Smaller instabilities are handled by the walking algorithm, which will stop walking for a moment to regain stability. There are currently no capture steps implemented. The detection if the robot is falling is a classification based on its current torso rotation and angular velocity, using data from the IMU. If a threshold in one direction is exceeded, the robot is declared as falling and the HCM will bring the robot in a safe position, depending on the direction of the fall. In this position, the robot will land on its elastic printed bumpers and therefore survive the fall without damage.

V. Evaluation

The HCM (including earlier versions) was successfully used by the team Hamburg Bit-Bots in three RoboCup Soccer world championships. During this time, the skills, e. g. the kick, were developed further and replaced by newer approaches. This worked well since the HCM allows the skills to be distinct ROS nodes. This shows the advantage in maintainability in comparison with monolithic approaches.

The soccer domain adds additional requirements that are well fulfilled by the HCM. The robot can be penalized via WiFi by the referee, which is implemented by invoking the stop function. Furthermore, the robot can be picked up suddenly during the game and should not move to prevent harm to humans. Due to the HCM, the high-level soccer behavior is completely independent and can focus on tactics.

One downside of the HCM is the introduced latency for motor goals which come from passing all goals through the HCM. We evaluated how much time is introduced, and the results can be seen in Figure 5. There is no additional delay introduced to the sensor feedback since this data is directly transmitted via ROS topics to all nodes.

When the robot gets an impact, it takes around 1 s before hitting the ground, depending on the applied force (see Figure 4). The robot needs 0.2 s to go into a safe position, leaving around 0.8 s time for the HCM to invoke the fall protection. The detection time varies with the magnitude of the impact, since stronger pushes can be detected by the introduced acceleration, while slower tilting is recognized by the robots orientation. For our platform, the HCM is able to detect the fall early enough to move the robot to a safe pose. Afterward, it will get the robot back to standing by using a keyframe animation that takes 10 s.

The standard ROS navigation stack with move base runs successfully on the robot. It is possible to give the robot a goal position (either by high-level behavior or teleoperation) and the robot will go to the position and meanwhile handle problems, e. g. falling, autonomously. Teleoperation also works by giving desired walk velocities, e.g. by using a game controller. This allows a more dynamic and direct control of the robot. It is useful for situations where precise navigation is needed or where the enjoyment of the operator is important. It is important to set acceleration and velocity limits when using direct control, as untrained operators will otherwise produce numerous falls.

VI. Conclusion and Further Work

In this paper, we presented an architectural approach that allows to control a humanoid robot as simple as a wheeled robot. An exemplary implementation was done for the RoboCup soccer domain, where it performed well.

The introduced latency could be improved by using ROS 2 which has a lower latency message transfer. The fall handling could be improved by using capture steps to prevent falls. It will be evaluated if the HCM can also be used for reinforcement learning on an actual robot. In this case, the HCM could ensure that the robot does not get damaged as well as provide a negative reward for falls.

The source code is available at:
https://github.com/bit-bots/bitbots_motion

References

[1] Marc Bestmann, Hendrik Brandt, Timon Engelke, Niklas Fiedler, Alexander Gabel, Jasper Güldenstein, Jonas Hagge, Judith Hartfill, Tom Lorenz, Tanja Heuer, et al. Hamburg bit-bots and wf wolves team description for robocup 2019 humanoid kidsize. Technical report, Universität Hamburg, 2019.

[2] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE journal on robotics and automation*, 2(1):14–23, 1986.

[3] Sachin Chitta et al. ros_control: A Generic and Simple Control Framework for ROS. *The Journal of Open Source Software*, 2017.

[4] Klaus Dorer, Ulrich Hochberg, and Michael Wülker. The sweaty 2019 robocup humanoid adultsize team description. Technical report, University of Applied Sciences Offenburg, 2019.

[5] Grzegorz Ficht, Philipp Allgeuer, Hafez Farazi, and Sven Behnke. Nimbro-op2: Grown-up 3d printed open humanoid platform for research. In *IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 669–675, 2017.

[6] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1), 1997.

[7] Kiyoshi Fujiwara, Fumio Kanehiro, Shuuji Kajita, Kenji Kaneko, Kazuhito Yokoi, and Hirohisa Hirukawa. Ukemi: Falling motion control to minimize damage to biped humanoid robot. In *IEEE/RSJ international conference on Intelligent robots and systems*, volume 3, pages 2521–2526. IEEE, 2002.

[8] Trent Houliston, Jake Fountain, Yuqing Lin, Alexandre Mendes, Mitchell Metcalfe, Josiah Walker, and Stephan K Chalup. Nuclear: A loosely coupled software architecture for humanoid robot systems. *Frontiers in Robotics and AI*, 3:20, 2016.

[9] Trent James Houliston. *Software Architecture and Computer Vision for Resource Constrained Robotics*. PhD thesis, The University of Newcastle, Australia, 2018.

[10] Dimitrios Kanoulas, Jinoh Lee, Abderrahmane Kheddar, and Yohei Kakiuchi. Humanoid robot falling: Fall detection, damage prevention, and recovery actions. In *EEE International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[11] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup. In *Proceedings of the first international conference on Autonomous agents*. ACM Press, 1997.

[12] Jason Kulk and James S Welsh. A nuplatform for software on articulated mobile robots. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2011.

[13] Leandro Tomé Martins, Christopher A Arend Tatsch, Eduardo Henrique Maciel, Reinhard Gerndt, and Rodrigo da Silva Guerra. A Polyurethane-Based Compliant Element for Upgrading Conventional Servos into Series Elastic Actuators. *IFAC-PapersOnLine*, 48(19):112–117, 2015.

[14] Marcell Missura, Maren Bennewitz, and Sven Behnke. Capture steps: Robust walking for humanoid robots. *International Journal of Humanoid Robotics*, 2020.

[15] Dragomir N Nenchev and Akinori Nishio. Ankle and hip strategies for balance recovery of a biped subjected to an impact. *Robotica*, 26(5):643–653, 2008.

[16] R Peter Bonasso, R James Firby, Erann Gat, David Kortenkamp, David P Miller, and Mark G Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):237–256, 1997.

[17] Martin Poppinga and Marc Bestmann. Dsd - dynamic stack decider: A lightweight decision making framework for robots and software agents. *in peer-review process*, 2020. doi:10.13140/RG.2.2.14585.01129.

[18] Jerry Pratt, John Carff, Sergey Drakunov, and Ambarish Goswami. Capture point: A step toward humanoid push recovery. In *6th IEEE-RAS international conference on humanoid robots*, pages 200–207. IEEE, 2006.

[19] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[20] Reimund Renner and Sven Behnke. Instability detection and fall avoidance for a humanoid using attitude sensors and reflexes. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2967–2973. IEEE, 2006.

[21] Aulia Khilmi Rizgi, Ryan Satria Wijaya, Ilham Fakhrul Arifin, Mochamad Ayuf Basthomi, Cipta Priambodo, Rokhmat Febrianto, Ibrahim Musthofainal Akhyar, Miftahul Anwar, Anhar Risnumawan, and Achmad Subhan Khalilullah. Eros - team description paper for humanoid kidsize league, robocup 2019. Technical report, Electronic Engineering Polytechnic Institute of Surabaya, 2019.

[22] Max Schwarz, Julio Pastrana, Philipp Allgeuer, Michael Schreiber, Sebastian Schueller, Marcell Missura, and Sven Behnke. Humanoid teensize open platform nimbro-op. In *Robot Soccer World Cup*, pages 568–575. Springer, 2013.

[23] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer, 2016.

[24] Seung-Joon Yi, Stephen G McGill, Larry Vadakedathu, Qin He, Inyong Ha, Jeakweon Han, Hyunjong Song, Michael Rouleau, Byoung-Tak Zhang, Dennis Hong, et al. Team thor's entry in the darpa robotics challenge trials 2013. *Journal of Field Robotics*, 32(3), 2015.

Acknowledgments: Thanks to the RoboCup team Hamburg Bit-Bots. Especially to Martin Poppinga, Timon Engelke and Finn-Thorben Sell for their work with the DSD, and to Florian Vahl for improving the fall detection.