A Role-based Language for Collaborative Robot Applications

Sebastian Götz, Max Leuthäuser, Jan Reimann, Julia Schroeter,
Christian Wende, Claas Wilke, and Uwe Aßmann

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
sebastian.goetz@acm.org, max.leuthaeuser@googlemail.com, {jan.reimann, julia.schroeter, c.wende, claas.wilke, uwe.assmann}@tu-dresden.de

Abstract. The recent progress in robotic hard- and software motivates novel, collaborative robot applications, where multiple robots jointly accomplish complex tasks like surveillance or rescue scenarios. Such applications impose two basic challenges: (1) the complexity of specifying collaborative behavior and (2) the need for a flexible and lightweight communication infrastructure for mobile robot teams. To address these challenges, we introduce NaoText, a role-based domain-specific language for specifying collaborative robot applications. It contributes dedicated abstractions to conveniently structure and implement collaborative behavior and thus, addresses the complexity challenge. To evaluate NaoText specifications, we introduce an interpreter architecture that is based on representational state transfer (REST) as a lightweight and flexible infrastructure for communication among robot teams. We exemplify the application of NaoText using an illustrative example of robots collaborating in a soccer game and discuss benefits and challenges for our approach compared to state-of-the-art in robot programming.

1 Introduction

Recent progress in robotic hard- and software has led to more sophisticated and easier programmable robot platforms. In addition to their classical domains in fabrication and research, robots are expected to become affordable for all-day applications within the coming decade, e.g., in home entertainment or facility management. This leads to new application scenarios where robots are distributed, mobile, and communicating, solving complex tasks in collaborating teams, e.g., rescue or surveillance missions.

Typically, robot behavior is implemented in one of two different ways: (1) general purpose languages (GPLs) such as C, C++, or Java can be used for robot programming or (2) abstract domain-specific languages (DSLs) (e.g., Microsoft's visual programming language (VPL), Choregraphe[1] from Aldebaran, or

[1] http://www.aldebaran-robotics.com/en/Discover-NAO/Software/choregraphe.html

LabView for Mindstorms NXT[2]) can be used and have gained momentum in the robotics community [8]. However, implementing collaborative scenarios is still a challenging task, as the declarative expression of collaborations as first-class programming constructs is not supported by any of these languages.

For the future of robot programming, we envision a role-based [23] approach to express and control robot collaborations. Achieving this vision induces two main requirements: First, abstraction from robot-platform specific concepts and support for the expression of collaborative robot applications in an easy and comprehensive way. Second, a flexible and lightweight communication infrastructure to enable execution and coordination of collaborative tasks among robot teams. We argue that such coordination can be realized using a (possibly external) controller that executes a given collaboration specification and monitors and steers collaborative robot behavior (cf. Fig. 1).

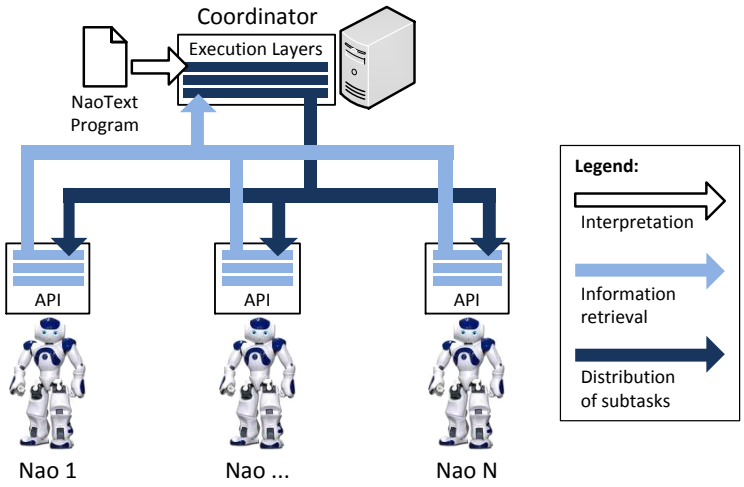

Fig. 1. Naos Working in Collaboration via a central Coordinator.

To address these requirements, this paper introduces NaoText, a role-based DSL for programming collaborative robot applications. NaoText is based on the concept of *contexts* allowing convenient specifications of how the entities of a system behave in respective contexts. This partial (i.e., context-dependent) behavior is denoted by *roles*, which interact to achieve collaborative behavior. As depicted in Fig. 1, NaoText is executed by a Java-based interpreter that is running on a central coordinator scheduling tasks of collaborative applications to several robots communicating with the coordinator. The communication is based on a lightweight, representational state transfer (REST)-ful service-oriented architecture (SOA) that consists of and interconnects two stacks of layers. The

[2] http://www.ni.com/academic/mindstorms/

first stack runs on the coordinator and evaluates a NaoText-based collaboration specification. The second stack of layers runs on each robot and exposes its functionality using REST-ful Web services that are accessed and invoked by the coordinator. This paper introduces both NaoText and its underlying implementation architecture. Furthermore, we exemplify the application of NaoText using a soccer game as an exemplary collaborative robot application and discuss it's advantages and disadvantages.

The remainder of the paper is structured as follows. In Sect. 2 we present our exemplary robot soccer application. In Sect. 3, we shortly outline the architecture implemented to evaluate NaoText projects. Sect. 4 introduces the foundations of NaoText for specifying collaborative robot applications by using the introduced example. We discuss benefits of and challenges for our approach in Sect. 5. A detailed discussion of related work follows in Sect. 6. Finally, Sect. 7 concludes this paper and outlines future work.

2 Motivating Example

To illustrate the need for a language which allows to express collaborations between robots, we introduce a motivating example in this section. A well-known testbed for the robotics community and interdisciplinary research is the RoboCup.[3] Robot soccer—an area of competition amongst others at the RoboCup—is a highly dynamic, collaborative, and context-sensitive game. Hence, for our motivating example, we choose a robot soccer scenario with Nao robots. The Nao—developed by Aldebaran Robotics—is a humanoid robot produced in series and offering a standard platform. It has been chosen as the model for the standard platform league at the RoboCup in 2011.

The basic structure of robot soccer is analog to usual soccer. Two teams play against each other and a referee takes care that every participant respects the rules. Each team comprises several roles: goalkeepers, strikers, defensive players, and sweepers. Depending on the number of robots per team,[4] only some of these roles are bound to players during a match. Further, a single role can be played by multiple robots (e.g., the defensive player role) or can be bound to only one robot per team (e.g., the goalkeeper role).

Notably, several contexts can be identified for a soccer match. Besides each team forming a separate context, situations at runtime determine contexts, too. If, for example, two players intend to pass the ball from one to the other, this pass defines a context of its own. Further examples include the shot on goal, the tackling, and the corner kick, to name but a few. Each context defines roles, which in turn specify how the participants in this context behave. For example, a robot, which intends to pass the ball to a teammate, starts playing the `Sender` role in the `Pass` context. It has to compute the angle to the receiving robot (`Receiver`) and shoot the ball in case there are no opponents in the trajectory. Fig. 2 depicts the central concepts used in our example. It first introduces the

[3] http://www.robocup.org/

[4] In robot soccer, less than eleven participants per team are common.

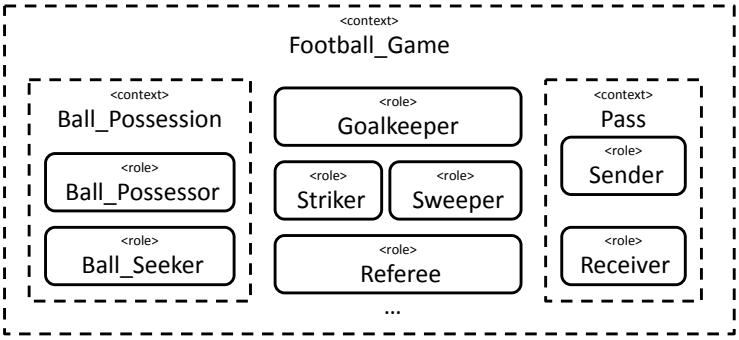

Fig. 2. Selected Concepts of Robot Soccer.

context `Football_Game` defining the above mentioned roles of a football team. In an inner context named `Pass` the special roles of players taking part in the pass collaboration (`Sender`, `Receiver`) are introduced.

Most of the logic behind soccer depends on the current game situation (i.e., on the current context of the collaborating robots). In a naive implementation, contexts and roles need to be represented using concepts of a conventional GPL. Here, we experience tangling code of different collaborations (i.e., behavior to be performed in different situations) and scattered code for a single collaboration across the code of the application by replication of if-statements.

Listing 1 shows a simplified code snippet, specifying the behavior for a robot. The behavior of the robot depends on whether it is in possession of the ball (`if (BALL_POSSESSION)`), whether it is in the role of a goalkeeper (`if (GOALKEEPER)`) and whether it is the sender or receiver of a pass (`if (SENDER)`, `if (RECEIVER)`). The checks for the ball possessor, sender and receiver roles

```
1  if (GOALKEEPER) {
2    if (BALL_POSSESSION) {
3      if(SENDER) { throw_ball: nearest_free_player: this; }
4      else if(RECEIVER) {...}
5    }
6    else {...}
7  }
8  else {
9    if(BALL_POSSESSION) { /* replicated if structure */
10     if(SENDER) { shoot_ball: nearest_free_player: this; }
11     else if(RECEIVER) {...}
12   } else {...}
13   ...
14 }
```

Listing 1. Example Behavior Specification Including four Roles.

need to be replicated in the else-branch of the check for the goalkeeper role. The illustrated nesting is required, because the actual behavior in the branches could differ. As shown in the example, a goalkeeper will throw the ball to the nearest player relative to himself (cf. line 3), whereas a usual player has to shoot the ball (cf. line 10). Notably, each additional role and each additional context will further impair replication. Scattering leads to issues in code maintenance, as a change in the behavior of a goalkeeper requires adjustments of multiple—syntactically distributed and unrelated—code segments. Hence, suitable abstraction and modularization mechanisms are required to avoid code scattering and replication. Before we present NaoText in Sect. 4—which comprises such mechanisms—we will outline our architecture in the next section.

3 Applying SOA for Simple Robot Coordination

This section introduces our architecture for implementing the interpreter used to evaluate NaoText-based specifications of collaborative robot applications. Shifting the focus of robot programming from singular automation units to distributed, mobile teams of collaborating robots induces the need for a communication architecture that establishes communication channels among the involved robots [26]. We suggest a distributed architecture that uses Web services for communication. The numerous benefits of Web services and SOA introduced in [12] are beneficial for the following reasons:

Platform Independence Web services provide a standardized communication protocol that is implemented in and can be accessed from various platforms. This is beneficial both for implementing services and the service interface for concrete robot platforms to remotely access and control these services.

Declarative Interface Web services declare an interface abstracting the physical implementation of a given functionality. This means implementation can be exchanged, adopted or ported to other infrastructures. This is beneficial w.r.t. the heterogeneity found in today's robot platforms.

Location Transparency Services are invoked through a communication network that routes service calls to the receivers independent of their location.

Various authors [14,16,18,24,25] suggest the implementation of SOAs for robot communication using the simple object access protocol (SOAP) or the common object request broker architecture (CORBA). Both approaches are often criticized for requiring a sophisticated messaging infrastructure. This is a problem, especially w.r.t. the restricted capabilities of software running on robots. We address this issue by implementing SOA using REST. As REST builds on HTTP and URIs for routing and calling Web services, it alleviates the need for custom communication middleware and can easily be deployed for robots connected through a wireless network. In the following, we describe the implementation of a REST-ful SOA for Nao robots.

Fig. 3 shows the application of SOA concepts to remotely control and coordinate a set of Nao humanoid robots. It consists of two stacks of layers. The

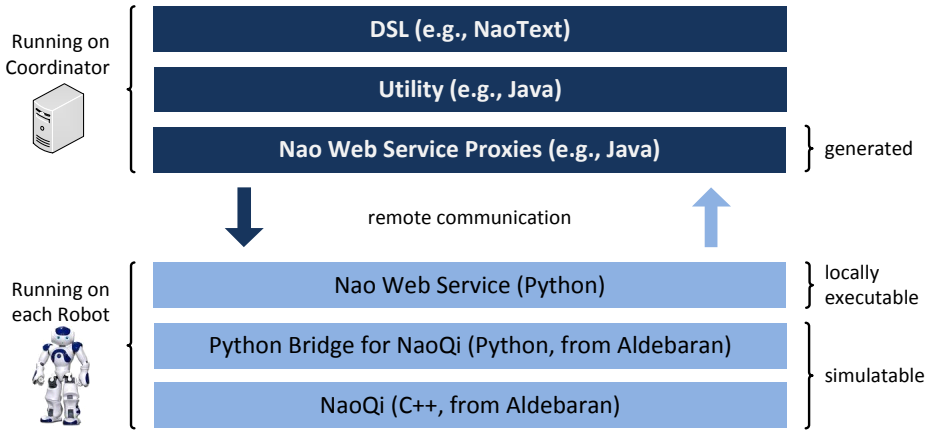

Fig. 3. Example for a layered architecture to control Nao robots.

lower stack contributes three layers implementing REST for Nao robots. The upper stack of layers runs on the *coordinator* that accesses this REST interface for controlling a collaborating team of robots (cf. Fig. 1).

Naos are controlled via NaoQi which represents the lowest layer of our architecture that is closest to the Naos' hardware (cf. Fig. 3). NaoQi provides an API for interacting with individual Nao sensors. Furthermore, it provides an API for basic actions such as controlling specific motors of a Nao or invoking the text-to-speech module. NaoQi is implemented in C++ and provided by Aldebaran. On top of NaoQi, Aldebaran provides a Python bridge that exposes the NaoQi API for the Python language. Using Python, we built a Web service layer that exposes the NaoQi services as REST-ful Web services. This Web service layer is automatically generated from the Python bridge using the introspection capabilities of Python. This approach strongly reduced our implementation effort and also eases the evolution of the Web service interface when NaoQi changes. Besides directly accessing the Web services, all provided services can be explored using a web browser. Each Web service can be called using HTML forms (generated by the Web service using templates). Our implementation is published[5] under GPL.

To access the Web service from the coordinator, we implemented a code generator that generates Java proxies. The proxies encapsulate the remote communication with the generated Web services. Above this layer, we implemented a number of utility interfaces that combine a number of low level Web services to more abstract behavior units. E.g., it provides a method `walkTo(int x, int y)` to let a Nao walk into a certain direction. Internally, the method computes the angle into which the robot has to rotate and how far it has to walk. Afterwards, a proxy object is created and the command is transmitted to the robot. Further above, in the top layer of our architecture, we find the interpreter for NaoText,

[5] `http://code.google.com/p/naoservice/`

which is described in more detail in Sect. 4. Briefly, the interpreter is executed by the coordinator and interprets NaoText programs. The interpreter also monitors the system status and manages the activation of contexts and bindings of roles to robots.

This simple SOA provides a communication infrastructure between our interpreter and the individual Nao robots. Similar solutions are easy to implement for other robot platforms. Further, the three lower layers can be simulated on desktop PCs to emulate the Nao robots. Thus, our architecture allows in-the-loop development and test of NaoText programs, too.

4 NaoText for Controlling Collaborating Robots

In this section, we introduce NaoText—a role-based DSL to specify the behavior of collaborating robots. It contributes an appropriate abstraction and ensures platform-independence for collaborative robot applications. We introduce the design of NaoText and illustrate its application on our motivational example.

4.1 Design of NaoText

In Fig. 4 we illustrate the design of NaoText by an excerpt of its metamodel. To manage the complexity of collaborative tasks we choose `Role` and `Context` as central abstractions in NaoText. The concept of roles, first applied over 30 years ago [2], embodies partial behavior of participants in a collaboration [19]. Thus, these concepts allow for concise specifications of collaborations. Roles are of founded, non-rigid types [15]. The property of being founded connotes that a type is dependent on another type. Consequently, `Role`s cannot exist on their own, but have to be contained in a `Context`. Non-rigidity connotes that instances do not cease to exist, when they stop having a non-rigid type. For example, a concrete robot being a striker does not cease to exist, when it stops being a striker. In consequence, each role needs to be bound to a player (e.g., the robot in the aforementioned example).

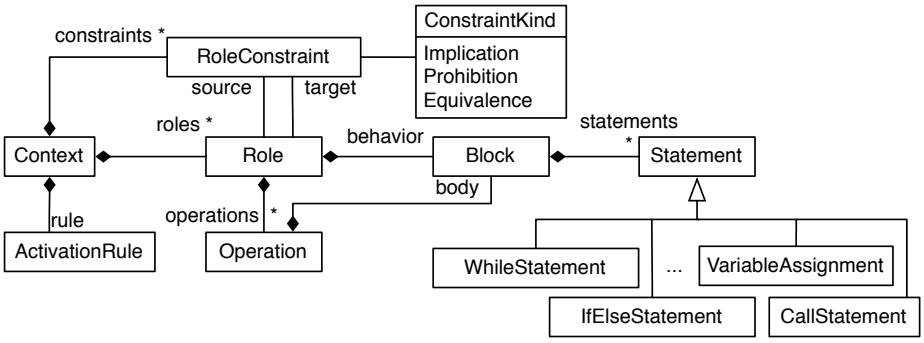

Fig. 4. Excerpt of Metamodel for the role-based NaoText DSL.

A multitude of approaches to constrain this binding exist. Besides hard constraints, restricting the players of a role to a single type as in ObjectTeams [17], complex set constraints have been investigated in the context of first-class relationships [5]. NaoText uses a lean, but expressive way to constraint role bindings that was introduced in [21]. It allows the definition of three kinds of binary `RoleConstraint`s: `Implication`, `Equivalence`, and `Prohibition`. If a role A implies role B, every player of role A has to play role B in addition. The role-equivalence applies the implication in a bidirectional way. Finally, if a role A prohibits a role B, no player of A is allowed to play role B at the same time.

The binding of roles to players during runtime is controlled using `ActivationRules`. These rules describe patterns for potential role players and trigger the activation of a context if matching players are found. Such context activation imposes the binding or rebinding of the roles for the matched players and initiates the evaluation of their respective behavior.

To model the behavior of roles in NaoText, each `Role` contains a `behavior Block`. This `Block` contains a list of `Statement`s. The kinds of statements available in NaoText resemble those typically found in imperative, object-oriented languages (e.g., `WhileStatement`, `IfElseStatement`, `VariableAssignment`). Using `CallStatement`s, behavioral specifications can be modularized. `CallStatement`s can refer to `Operation`s or call Web services defined for Naos.

4.2 Application on our Motivating Example

In the following, we will present the application of NaoText for our motivating example. First, we discuss how the declarative part of NaoText (roles and contexts) is used to structure robot collaborations. Then, we present the application of imperative language concepts.

Listing 2 shows the declaration of the `Pass` (lines 11–44) context as introduced in Fig. 2 in NaoText syntax. The context `Pass` introduces two roles: the `Sender` (lines 24–36) and the `Receiver` (lines 38–43). The context's activation is controlled by the activation rule defined in lines 14–22. The rule relates to roles defined in the surrounding context `Ball_Possession` (lines 1–45). Whenever a robot playing the role `BallPossessor` identifies another robot playing the role `BallSeeker` and is itself not able to do a shot on goal, the context `Pass` is activated. As a result of the activation, the `BallPossessor` becomes a `Sender` and the `BallSeeker` becomes a `Receiver`. Multiple such rules can be defined for each context. In addition, a role-prohibition constraint expresses the exclusiveness of the roles `Sender` and `Receiver` (line 12).

Besides the definition of contexts and roles, we use NaoText to describe the robots' behavior w.r.t. roles they are currently playing. Therefore, we use imperative language constructs to control services to be fulfilled by the robots. In Listing 2 both roles (`Sender` and `Receiver`) comprise a `behavior` block, which defines the roles' behavior (lines 26–34 and 39–42). The `Sender` role defines the behavior to be performed by a ball possessing robot, which intends to pass the ball. If the robot concludes that the ball is catchable by an opposing robot, it will feint a shoot and walk away. Else a pass is performed by passing the ball

```
1  context Ball_Possession {
2    role BallSeeker {
3      behavior {...}
4      void randomWalkWithBall {
5        walk_to: random_int(20), random_int(20)
6      }
7      //...
8    }
9    role BallPossessor {...}
10
11   context Pass {
12     Sender prohibits Receiver;
13
14     activate for {
15       BallPossessor p;
16       BallSeeker s;
17     when {
18       (p.robotInVision: s) and not (p as Striker).isGoalShotPossible;
19     } with bindings {
20       p->Sender;
21       s->Receiver;
22     }
23
24     role Sender {
25       attr passRatio: float;
26       behavior {
27         if(ballCatchableByOpponent) {
28           feintShoot;
29           randomWalkWithBall;
30         } else {
31           boolean hit = shootBall;
32           updatePassRatio: hit;
33         }
34       }
35       void updatePassRatio hit:boolean {...}
36     }
37
38     role Receiver {
39       behavior {
40         waitForBallInVision;
41         catchBall;
42       }
43     }
44   }
45 }
```

Listing 2. Example NaoText Code including the Pass Context.

to another robot. Notably, the `Sender` role includes an attribute for the ratio of successful or failed passes, which can be used for a more sophisticated decision making procedure than in the example shown here. The `Receiver` role defines that the robot will wait for the ball to appear in its vision followed by catching the ball. After the ball has been catched by the `Receiver`, the context is finished and other context activation rules will trigger a reconfiguration of role bindings. At runtime, the NaoText interpreter evaluates the description of role behavior and invokes the `behavior` method of the respective roles. The individual instructions are evaluated and, finally, transmitted as calls to the robot playing the respective role using the communication infrastructure introduced in Sect. 3.

In this section, we applied NaoText to declare roles and contexts and to imperatively describe the roles' behavior for soccer playing. Of course, implementing a complete soccer scenario involves more contexts and roles as outlined above. Also the concrete behavior specification are likely to be more complex as well. However, we think that the example can give an impression of how NaoText can improve the development of collaborative robot applications. In the next section, we present an initial evaluation of NaoText by discussing its benefits and challenges w.r.t. alternative approaches to specify robot collaborations.

5 Discussion

Using a SOA comes together with both advantages and disadvantages in the context of collaborative robot applications. On the one hand, using a SOA instead of other mechanisms like remote method invocation (RMI), leads to communication overhead at runtime. Thus, we decided to use a REST-ful SOA to minimize this communication overhead. On the other hand, SOAs allow for location transparency and uniform addressability. Especially, our proxy generator for the Nao Web service provides easy connectivity for every GPL.

As our approach proposes the execution of NaoText programs on a central coordinator node, it can be considered as an approach that shifts the application logic from individual robots into the cloud. This leads to thin robots that propagate their state to the coordinator node and receive commands they have to execute to interact with their environment. This approach has advantages as well as disadvantages, too. Coordinating all participating robots from a central node leads to operability problems once the coordinator node fails. In this situation, the robots can neither communicate with each other nor collaborate. However—when operating—the central coordinator node maintains the global state of the application and thus, knows about the robots' locations and their further attributes (e.g., whether or not they are close to other robots etc.). Thus, the coordinator can easily grant and remove roles from individual robots to control the collaboration in an optimal way. Furthermore, the deployment of the application logic into the cloud avoids the computation of complex applications on the robots. Thus, they can save their often limited resources for basic functionality which leads to better robot operability (e.g., longer battery life time).

In addition of using a central coordinator node, we propose to use a role-based DSL which leads to the following advantages and disadvantages: First, of course, the implementation of a new DSL imposes efforts such as the language's design and the implementation of a parser, an interpreter, and/or a compiler. However, afterwards, the DSL leads to several advantages in contrast to GPLs. A DSL allows for abstraction from general-purpose concepts and expressing the behavior in lesser instructions being more appropriate for the domain of the language (i.e., humanoid robots). For example, instead of writing a block of Java code comprising several statements to let a Nao walk to a certain position, a single statement is sufficient to express the same behavior within NaoText. Thus, DSLs lead to more intuitive behavior implementations that are easier to understand and maintain. Besides, by introducing imperative robot commands that abstract from interface invocations for robot-platform specific services, the DSL can be easily connected to other robots. Instead of writing a completely new NaoText program, we only have to create new service adapters for the newly introduced robot type (e.g., to integrate Mindstorm robots into our soccer scenario). Besides the advantages of DSLs in general, the role-based concepts integrated into NaoText allow for a declarative expression of robot collaborations. Code replication and sections of nested if-statements can be avoided. By this, the utilization of roles fosters comprehensibility and maintainability. A similar implementation in imperative source-code would lead to a robot's behavior scattered over many if-statements checking its current contexts and state.

Finally, the annotation and analysis of non-functional properties is vital for handling safety-, time-, or resource critical characteristics of robots interacting with the physical world. A DSL offers the right level of abstraction to easily implement static analysis and testing capabilities. E.g., role constraints provide an easy concept to prohibit behavior of robots that is expected to be not executed in parallel (e.g., a referee should not shoot a goal). Furthermore, the imperative statements of NaoText provide the right level of abstraction for the estimation of non-functional properties (e.g., execution time or energy consumption) of individual Naos, whereas by using a complete GPL for static analysis, large parts of the analysis would have to deal with several general-purpose concepts being less important for the context of collaborative robot applications.

6 Related Work

NaoText as presented in this paper contributes a communication architecture and a role-based DSL to specify the behavior of collaborative robot applications. In this section we discuss related work and differences to our approach.

Communication Aspects. The authors of [26] state that communication between robots is essential for team play and therefore discuss explicit communication as an approach for robot collaborations. Many works use SOA to address the problem of realizing the collaboration between robots. The works of [14,16,25] propose either to use CORBA or SOAP for the communication among robot teams.

In contrast, we decided to use REST-ful services—like some platforms for networked mobile robotics do [6,11]—because REST does not need any additional communication middleware as it has no additional transport layer. Furthermore, [24] proposes to use Web services to access and control robots, where a single machine orchestrates the synchronization between different tasks and robots. This infrastructure offers control over a group of heterogeneous robots through the Internet. Thus, they address the lack of standardized interfaces and communication protocols to interconnect heterogeneous robots over the Internet. In contrast to our approach, the Web service protocol was not implemented directly on the robots and no DSL to specify collaborations is presented. Chen and Bai describe another scenario with collaborative robots [8]. In their approach the robots are orchestrated by a remote collaboration center (RCC) and communicate via Web services. The *Robotics DeveloperStudio* from Microsoft is used to realize collaborations. To ease the communication between robots and to provide reusable software components, the *robot operating system (ROS)* framework has been developed [18]. It provides a communication layer for robot applications that can be used for both peer-to-peer communication and SOA-like communication between nodes in robot applications. Besides the communication layer, ROS provides a hardware abstraction layer and libraries for commonly used functionality. The framework is language independent in the way that it is available for multiple GPLs, like Phyton, C++ and Lisp. Experimental implementations for Java and Lua exist as well. The open-source framework *urbi*[6] can be used to control robots and their collaboration in general. Therefore it offers a low level C++ component library, which simplifies the process of writing programs. Multiple robot-specific implementations of urbi exists. As the resources for computing algorithms on robots themselves are limited, recent efforts have been spent to shift computationally intensive functions into the cloud. *DAvinCi* is such a cloud computing framework for collaborating service robots [1]. In DAvinci robotic algorithms are exposed as a service. The data is shared among the robot cluster cooperatively. The term *robot as a service (RaaS)* is created in [9]. The work shows, that a robot can be used in the cloud as a SOA-unit, providing and consuming services and acting as a service broker. Challenges for the development of distributed robotic services are summarized in [20].

Language Aspects. Using a DSL to describe robot behavior and collaboration is promising, since it helps to abstract from hardware and low level implementation details. *Spica* is a model-driven software development environment for creating multi-robot communication infrastructures [3,4]. The development environment consists of several DSLs to specify the different aspects of robot communication. In contrast to our approach, the DSL used to describe robotics' behavior is not capable of roles. Naos are delivered together with the development environment *Choregraphe*. It allows developing a Nao's application in a graphical and component-based manner. Complex applications can be plugged together using basic building blocks like *stand up*, *sit down*, or *text-to-speech*. Although

[6] `http://www.urbiforge.org/`

Choregraphe allows easy development of applications for single Nao robots, it does not support Nao interaction nor interaction between Naos and other robots or players in its environment, neither in an imperative nor in a declarative way. The *Robotics DeveloperStudio* from Microsoft provides a runtime environment to create, host, manage, and connect distributed services. The language used in the Robotics DeveloperStudio is called *VPL* . It builds on the .NET framework and is comparable to the language used in Choregraphe.

Roles for Collaborative Robot Applications. In our approach, we propose to use roles to describe the collaboration between robots. The applicability of roles in controlling cooperative robots is examined in [7]. The robots act independently, but can react to messages they send to each other. The concept of roles as used in [7] is comparable to a state the robot is in at a time. Therefore, roles and role changes are expressed as a finite automaton for each robot. Our example introduced in Sect. 2 shows the ability of a robot to play multiple roles at once. In consequence, modeling role changes as transitions in an automaton leads to an exploding number of states. This is because each possible combination of roles has to be represented by a separate state of the automaton. In contrast, we declare role and context changes by means of activation rules as shown in Listing 2. An approach using roles for the collaboration of modules a robot consists of is presented in [22]. In this approach, roles are used to express the (active and reactive) behavior as well as the structure of the modules of a robot. The authors provide a role-based DSL called *Role-based Distribution Control Diffusion (RDCD)* to implement their approach for the ATRON self-reconfigurable robot. The language is an extension to *RAPL (Role-based ATRON Programming Language)*, that is presented in [10]. Role changes can occur as a reaction of a message from one of the robots' modules or internal events send by sensors of a robot. The DSL provides primitives for making simple decisions, whereas complex computations are described externally. They focus on a single robot and the collaboration between its modules, where only neighbor modules are able to collaborate. In contrast, our approach focuses on the collaboration of multiple independent robots.

7 Conclusion

In this paper we discussed the fundamental concepts of NaoText, a DSL supporting role concepts for the implementation of collaborative robot applications. We presented an architecture to evaluate NaoText programs based on a central coordinator that monitors the system status and controls collaborative robot teams. For communication between the coordinator and the individual robots this architecture uses REST-ful Web services. We demonstrated the application of NaoText using an exemplary football application. Our approach leverages abstraction and comprehensibility in the implementation of collaborative robot behavior and contributes a lightweight architecture for collaborations among robots that is easily extensible to support further robot platforms besides Naos. We discussed differences and commonalities of our approach with related work.

For future work, we plan to implement an interpreter for NaoText and to complement the NaoText tooling with static analyzers, to ensure functional and non-functional properties (NFPs) (e.g., performance and energy consumption) and to develop testing capabilities for NaoText applications. We also plan to improve the expressiveness of NaoText by integration of predicate dispatch [13] (i.e., the dynamic selection of appropriate method implementations w.r.t. predicates over the state, structure and NFPs of the system). Notably, the currently used role dispatch is a special case of predicate dispatch. Finally, we will extend our interpreter architecture to support further robot platforms and other physical devices, to enable the specification of sophisticated cyber-physical systems.

Acknowledgement This research is funded by the DFG within CRC 912 (HAEC), the European Social Fund and Federal State of Saxony within the project ZESSY #080951806 and by the European Social Fund, Federal State of Saxony and SAP AG within project #080949335.

References

1. R. Arumugam, V. Enti, L. Bingbing, W. Xiaojun, K. Baskaran, F. F. Kong, A. Kumar, K. D. Meng, and G. W. Kit. DAvinCi: A cloud computing framework for service robots. In *IEEE International Conference on Robotics and Automation (ICRA2010)*, pages 3084–3089, 2010.
2. C. Bachman and M. Daya. The role concept in data models. In *Proceedings of the 3rd Conference on Very Large Data Bases (VLDB)*, pages 464–476, 1977.
3. P. A. Baer and R. Reichle. *Communication and Collaboration in Heterogeneous Teams of Soccer Robots*, chapter 1, pages 1–28. Tech Education and Publishing, Vienna, Austria, 2007.
4. P. A. Baer, R. Reichle, and K. Geihs. The Spica Development Framework - Model-Driven Software Development for Autonomous Mobile Robots. In *Proceedings of International Conference on Intelligent Autonomous Systems (IAS'10)*, pages 211–220. IAS Society, IAS Society, 2008.
5. S. Balzer, T. R. Gross, and P. Eugster. A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP2007)*, pages 323–346, 2007.
6. E. Cardozo, E. G. Guimaraes, L. A. Rocha, R. S. Souza, F. Paolieri Neto, and F. Pinho. A platform for networked robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'10)*, pages 1000–1005, Taiwan, 2010.
7. L. Chaimowicz, M. Campos, and V. Kumar. Dynamic role assignment for cooperative robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA02)*, pages 293–298. IEEE, 2002.
8. Y. Chen and X. Bai. On Robotics Applications in Service-Oriented Architecture. In *28th International Conference on Distributed Computing Systems Workshops (ICDCS2008)*, pages 551–556, 2008.
9. Y. Chen, Z. Du, and M. Garcia-Acosta. Robot as a Service in Cloud Computing. *IEEE International Workshop on Service-Oriented System Engineering*, pages 151–158, 2010.

10. N. Dvinge, U. Schultz, and D. Christensen. Roles and Self-Reconfigurable Robots. *Roles07*, pages 17–26, 2007.
11. R. Edwards, L. E. Parker, and D. R. Resseguie. Robopedia: Leveraging Sensorpedia for Web-Enabled Robot Control. In *Proceedings of 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM2010)*, pages 183–188, 2010.
12. T. Erl. *Service-oriented architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.
13. M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP 1998*, pages 1860–211. Springer, 1988.
14. L. Flückiger, V. To, and H. Utz. Service-Oriented Robotic Architecture Supporting a Lunar Analog Test. In *International Symposium on Artificial Intelligence, Robotics, and Automation in Space (iSAIRAS)*, 2008.
15. N. Guarino and C. A. Welty. A Formal Ontology of Properties. In *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management (EKAW2000)*, pages 97–112, London, UK, 2000. Springer.
16. Y.-G. Ha, J.-C. Sohn, and Y.-J. Cho. Service-Oriented Integration of Networked Robots with Ubiquitous Sensors and Devices using the Semantic Web Services Technology. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005. (IROS2005)*, pages 3947–3952, 2005.
17. S. Herrmann. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.
18. M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.
19. T. Reenskaug, P. Wold, and O. Lehne. *Working with objects - The OOram Software Engineering Method*. TASKON, 1995.
20. S. L. Remy and M. B. Blake. Distributed Service-Oriented Robotics. *IEEE Internet Computing*, 15:70–74, 2011.
21. D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA1998)*, pages 117–133, New York, NY, USA, 1998. ACM.
22. U. P. Schultz, D. J. Christensen, and K. Stoy. A domain-specific language for programming self-reconfigurable robots. In *Workshop on Automatic Program Generation for Embedded Systems (APGES)*, pages 28–36, October 2007.
23. F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *IEEE Transactions on Data and Knowledge Engineering*, 35(1):83–106, 2000.
24. V. M. Trifa, C. M. Cianci, and D. Guinard. Dynamic Control of a Robotic Swarm using a Service-Oriented Architecture. In *13th International Symposium on Artificial Life and Robotics (AROB2008)*, 2008.
25. B. Wu, B.-H. Zhou, and L.-F. Xi. Remote multi-robot monitoring and control system based on MMS and web services. In *Industrial Robot: An International Journal*, volume 34, pages 225–239. 2007.
26. K. Yokota, K. Ozaki, N. Watanabe, A. Matsumoto, D. Koyama, T. Ishikawa, K. Kawabata, H. Kaetsu, and H. Asama. UTTORI United: Cooperative Team Play Based on Communication. In *RoboCup-98: Robot Soccer World Cup II*, pages 479–484, London, UK, 1999. Springer.