Modeling Reusable, Platform-Independent Robot Assembly Processes

Arvid Butting[1], Bernhard Rumpe[1], Christoph Schulze[1], Ulrike Thomas[2], and Andreas Wortmann[1]
[1] Software Engineering, RWTH Aachen University, Germany
[2] Robotics and Human-Machine Interaction, Technical University Chemnitz, Germany

***Abstract*— Smart factories that allow flexible production of highly individualized goods require flexible robots, usable in efficient assembly lines. Compliant robots can work safely in shared environments with domain experts, who have to program such robots easily for arbitrary tasks. We propose a new domain-specific language and toolchain for robot assembly tasks for compliant manipulators. With the LightRocks toolchain, assembly tasks are modeled on different levels of abstraction, allowing a separation of concerns between domain experts and robotics experts: externally provided, platform-independent assembly plans are instantiated by the domain experts using models of processes and tasks. Tasks are comprised of skills, which combine platform-specific action models provided by robotics experts. Thereby it supports a flexible production and re-use of modeling artifacts for various assembly processes.**

I. Introduction

Future smart factories require flexible production of highly individualized goods in small and medium lot sizes, where reconfigurations of systems occur with high frequencies. To achieve this, such factories require flexibly usable assembly line robots, which can work safely in shared environments with humans. Flexible usage of such robots requires easy programming for arbitrary tasks. Compliant manipulators allow such interaction, but programming these robots is more complex than programming rigid industrial robots. Beside knowledge of a general-purpose programming language (GPL), it also requires control specific knowledge to adjust compliance parameters like stiffness and damping for each motion. Thus, only robotics experts with sufficient programming knowledge are able to program such compliant manipulators. Furthermore, the re-usability of such control specific programs is endangered due to their target specific nature. Making these usable and re-usable in daily flexible production requires tools for a more abstract development of assembly tasks that are executable by shop floor workers with none software engineering expertise.

In [1] we propose a new framework which consists of a domain-specific language (DSL) and a toolchain to generate specialized robot programs for assembly tasks. The recent robot programming interface for the iiwa-LBR is based on the compliance-frame concept introduced by Mason and further developed with the task-frame-formalism by [2]. The programming interface uses stiffness and damping definitions for each Cartesian DOF while the robot moves towards a target pose. Stop conditions can be applied in order to immediately stopping the motion and awaiting new specifications for the next motion. Fig. 1 illustrates the programming specification necessary for the action shown at the right side of the figure. The robot rotates about the x-axis in the task-frame, while it pushes the object towards the top hat rail until the torque about the x-axis exceeds a certain threshold. For Cartesian force controlled robots the skill-primitive concept has been suggested earlier [3], [4], which is also based on the task-frame formalism. Skill primitives (SPs) are combined to skill primitive nets [5], [6], [7] (SPNs) and their transitions are ensured by preconditions and postconditions. Figure 2 shows a SPN consisting of three SPs, which describes the placement of an object onto a table. The surface orientation of the plane is not precisely known. Hence, a force-torque sensor is attached to the robot hand flange. Therewith contact forces and torques can be measured and evaluated during the assembly process. According to contact forces and torques, different SPs are selected for execution. In this example, either a rotation along the object's depths axis or its width axis is carried out. When the stop conditions trigger, a transition to one of the subsequent SPs, depending on the measured values of the force-torque sensor, is performed.

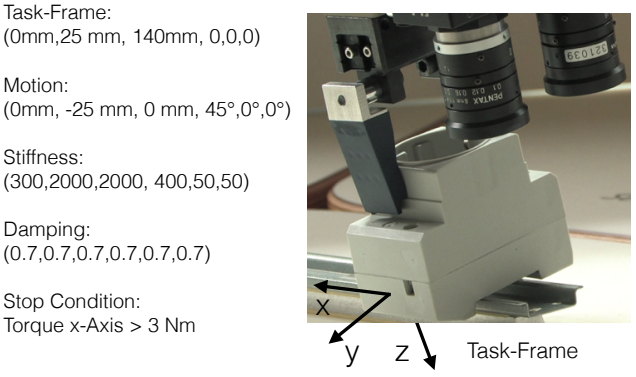

Fig. 1. Left side: Motion commands for the LBR-robot to establish the task. Right side: Position of the task frame for the rotation about the x-axis to assemble the socket onto the top hat rail.

SPNs have proven useful for the programming of complex robot tasks but lack abstraction, separation of concerns between shop floor workers and robotics experts. As it can be seen, the art of programming different robots in the assembly domain, might be the same. Hence a concept is necessary which is grounded on domain-specific languages

This research has partly received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n¡ 287787, SMErobotics.

[BRS+15] A. Butting, B. Rumpe, C. Schulze, U. Thomas, A. Wortmann
Modeling Reusable, Platform-Independent Robot Assembly Processes
In: Proceedings of the Sixth International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015), 2015
www.se-rwth.de/publications

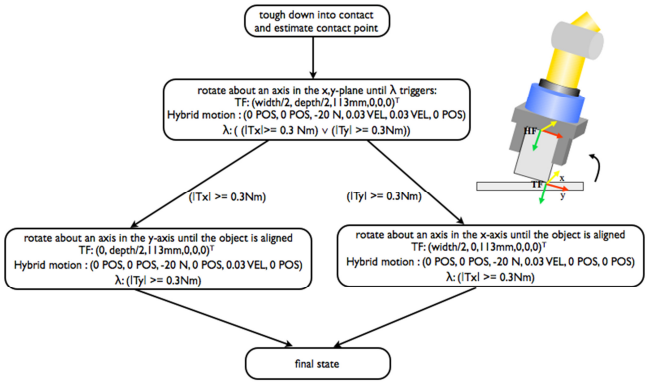

Fig. 2. Excerpt of a SPN for placing an object onto a table despite uncertainties about the orientation of table's plane.

and allows the programming of different robots in an intuitive way triggered by the problem domain rather than the robot hardware. It motivated us to provide such a framework which allows experts to exploit robotic hardware while non-experts can use their results in an intuitive way.

Figure 3 illustrates how robot assembly tasks are modeled as three-level networks with LightRocks [1] instead. We have adapted the concept of SPs such that it can be used for compliant manipulators. In order to increase the level of abstraction we distinguish between assembly *plans*, *processes*, *tasks*, *skills*, and *actions*: Assembly plans are provided externally [8] and consist of assembly processes. Each assembly process consists of tasks that consist of skills. Assembly processes and tasks can be modified by domain experts to adjust the assembly process to fit new environmental conditions, e.g., because the robot uses another gripper than assumed, a sensor does not work as expected, or workpieces are placed differently than the expert system assumed. Assembly skills consist of actions, which are platform specific and provided by robot experts.

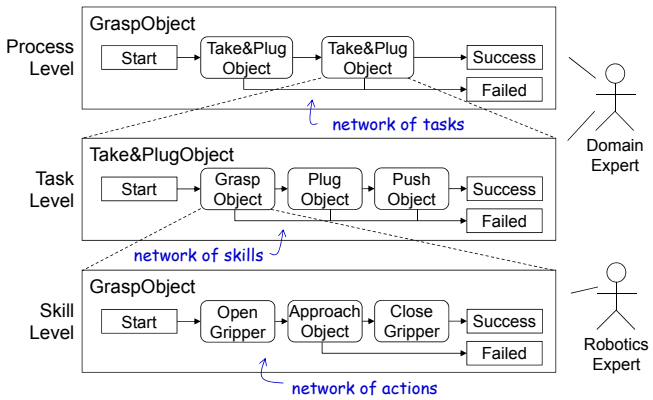

Fig. 3. The LightRocks abstraction layers.

These levels of abstraction allow to separate robotics expertise modeled within actions and skills from domain expertise embodied in tasks and processes. It also allows re-using recurring skills for different assembly tasks and re-curring tasks for different assembly processes. The previous version of LightRocks [1] was implemented as a profile of the UML/P [9] Statechart (SC) language with the MontiCore language workbench [10], [11]. The UML/P is a variant of UML [12] for programming. While this allows re-use of language infrastructure, description of domain types via UML/P class diagram (CD) models [13], model analyses, and code generators [14], the resulting modeling language is less comprehensible than intended and requires domain experts to comprehend the full expressiveness of UML/P SCs. To liberate domain experts from this, we present a collection of MontiCore DSLs for concise representation of SPNs to facilitate development of re-usable, platform-independent robot assembly tasks. In addition, the degree of re-usability for each introduced abstraction layer regarding different tasks, APIs, target platforms or robots is evaluated.

In the following, Sect. II illustrates LightRocks by example before Sect. III discusses related work. Sect. IV describes the new LightRocks DSLs and toolchain. Afterwards, Sect. V outlines case studies and finally, Sect. VI debates future work and summarizes the contribution.

II. Example

The assembly task of placing a screw into a thread may be composed by grasping the screw, moving it to the thread, and tightening it into the thread. Figure 4 depicts a part of the task `GraspAndScrew` that uses the gripper to pick up a screw and tightens it into a thread. The task consists of several skills and provides two outcomes: either the screw is placed accordingly to the skill `Screwing` or it is not. The skill `Screwing` inserts a screw into a thread after a previously executed skill placed it accordingly. It consists of four actions of which `Spin` and `CloseGripper` are illustrated in Fig. 4.

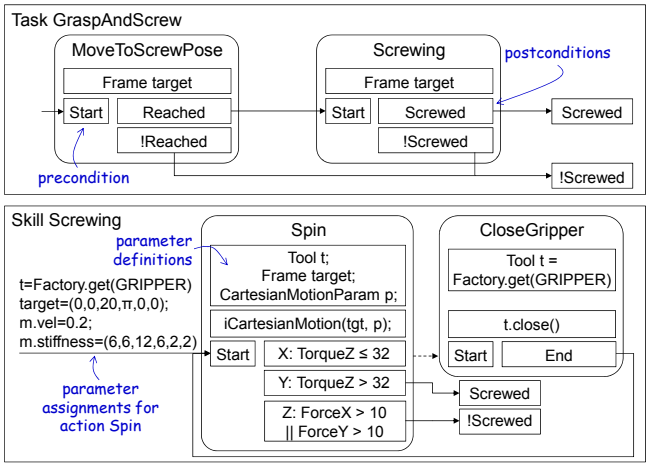

Fig. 4. Excerpts of task `GraspAndScrew` and skill `Screwing`.

The developer modeled the skill following human behavior: after initially grasping the screw and holding it over the thread, the robot spins the screw (action `Spin`). If a certain torque is reached, the screw is fixed and the skill finishes. Otherwise the robot releases the gripper, rotates back, grasps the screw again (cf. action `CloseGripper`), and spins it

again. Note that the action `Spin` yields two further outcomes to detect whether something besides the robot manipulated the workpiece. Such multiple outcomes allow modeling flexible skills that can deal with uncertainties. The action `CloseGripper` references the type `Tool`, which is part of the robot interface of the domain model, via `t.close()`.

LightRocks uses MontiCore to validate such models and to generate proper GPL implementations of these models. The resulting robot behavior is illustrated in Fig. 5.

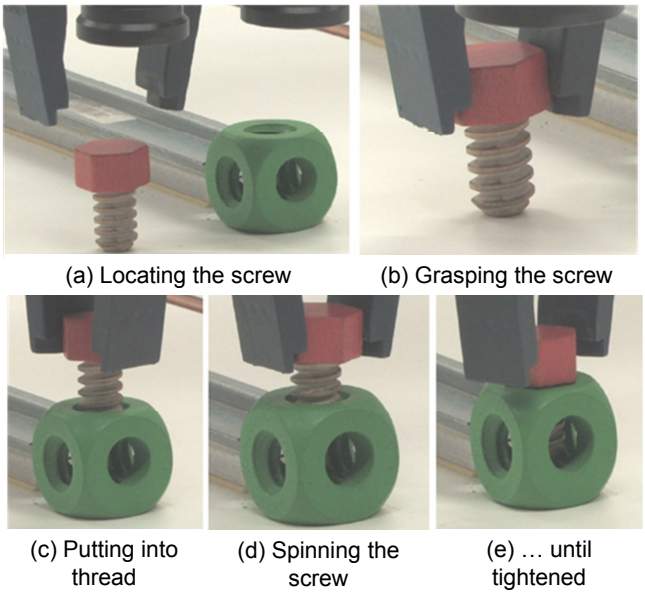

Fig. 5. A KUKA LBR robot finding and tightening a screw using LightRocks

III. Related Work

Recently, multiple DSLs for imperative or event-driven robot behavior [15], [16], [17], perception tasks [18], and software architecture with state-based behavior descriptions [19], [20], [21] have been presented. While not focused on assembly tasks, these behavior modeling languages are related in the common aim to facilitate robot programming.

Current approaches to robot behavior modeling aim at robotics software engineers, not domain experts. To this end, these approaches either provide less abstraction [15], [16], [17] or require knowledge on automata semantics to describe behavior [19], [20], [21]. Even previous LightRocks [1] and closely related approaches [22], [23] expect certain degrees of software engineering knowledge from the domain experts. With current LightRocks, this is relieved further as well-formedness rules prohibit tasks and skills to reference the robot's API. Thus, domain experts only need to comprehend task composition, skill composition and domain parameter assignment.

Rethink developed an industrial robot called Baxter which can be trained manually by ordinary line workers [24]. Different kind of tasks, like performing a blind pick or placing an object in a grid pattern, can be configured by interacting with the UI and teaching positions and areas by moving the robot's end effector directly. As described in subsection V-A LightRocks supports manual teaching of different poses, too. Unlike LightRocks the approach provides a user-friendly UI to define tasks only, while the proposed skill level is hidden to the end user and consequently new skills can not be defined by the customer directly.

IV. LightRocks Languages and Toolchain

LightRocks is developed as an integrated collection of MontiCore [11] languages which comprises of a process language, a task language, a skill language, and an action language. In this collection, each language comprises of (i) a context-free grammar (CFG) [25], (ii) well-formedness checks, to check properties not expressible with CFGs, and (iii) symbol tables [26], [13], which give information about imported models of the same and other languages to enable language integration. The latter, for instance, enable integration with UML/P CDs to reference the models representing domain types and type checking between models. Models of LightRocks are textual [27], which allows easy comprehension even of large models by software engineering experts, liberates these from layouting efforts, and enables processing models with their accustomed tools.

Processes and tasks represent the logical structure of reusable assembly process knowledge. To this effect, processes contain a net of tasks and tasks contain a net of skills. Both may refer only to domain types. Skills contain nets of actions. Actions, however, may reference a single method of the underlying domain model only. The domain model describes the interfaces of sensors, tools, the types of movement that can be used (e.g., linear or rotational), and the available domain types in terms of a UML/P CD language profile that restricts class diagrams to interfaces. Using different CDs for domain types and robot interfaces separates concerns and enables to re-use both with arbitrary manipulators, as long as a code generator from CD to the manipulator's GPL is present. Ultimately, it grounds the assembly processes via actions to the robot and allows validating the models.

We implemented previous LightRocks as a profile of UML/P SC where tasks, skills, and actions were handled uniformly as states. This led to "notational noise" [28], e.g., unintuitive language elements, and increased the "accidental complexity" [29] by forcing domain experts to learn SCs instead of using an assembly domain specific language. The new stand-alone LightRocks languages clearly separate between language elements for domain experts and language elements for robotics experts. This reduces noise and complexities. Listing 1 shows the textual model of the action `Spin` as depicted in Fig. 4.

The textual syntax is straightforward and consists of the keyword `action` followed by a name (l. 1), a `parameters` declaration block (ll. 3-7), which defines how the action can be parametrized, an `execution` block (ll. 9-11) that may reference the robot API, and a set of entry and exit rules (ll. 13-16) to define preconditions and postconditions of the action. Parameters are assigned via incoming

Action

```
action Spin {

  parameters {
    Tool t;
    Frame target;
    CartesianMotionParam p;
  }

  execution {
    iCartesianMotion(target,p);
  }

  entry Start;
  exit X: t.torqueZ()<=32;
  exit Y: t.torqueZ()>32;
  exit Z: t.forceX()>10 || t.forceY>10;
}
```

Listing 1. Model of the action `Spin` as show in Fig. 4.

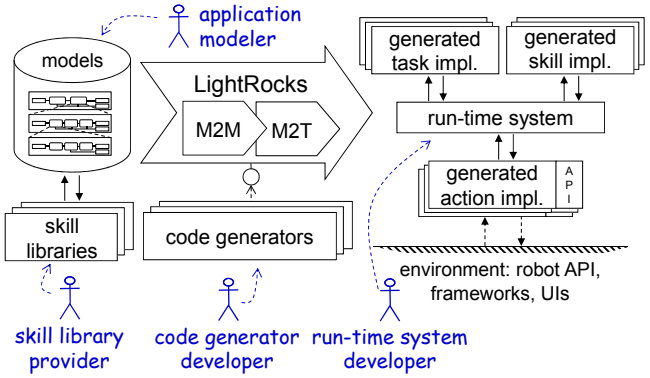

Fig. 6. The LightRocks toolchain translates assembly processes consisting of tasks, skills, and actions via Statecharts to executable code.

transitions and locally visible in task, skill, or action. Tasks and skills additionally propagate their parameters to the contained topology.

Based on the grammars of LightRocks languages, MontiCore generates language processing infrastructure including FreeMarker-based[1] code generation sub-frameworks and text editors [30]. The LightRocks toolchain utilizes these to parse models, process these, and ultimately transform these into executable code. Figure 6 depicts the toolchain with the related roles, according to [31].

As current and previous LightRocks are functionally equivalent, retaining compatibility with existing models, tooling, and code generators is straightforward: task, skill, and action models are transformed into SC representations compatible with previous LightRocks which can be processed by existing tooling. The current LightRocks toolchain parses process models provided by the *application modeler*. Robotics experts acting as *skill library providers* may provide the latter. Current code generators retain the structural separation into tasks, skills, and actions, which interact with a run-time system that, for instance, defines how to execute transitions. The LightRocks toolchain provides extension points for code generators to enable code generation for arbitrary target platforms with minimal effort. Code generators and run-time system (RTS) are not specific to a certain robot but to the programming language to be used and non-functional requirements (e.g., restrictions to the amount of memory to be used). Thus, *code generator developers* and *RTS developers* require software engineering expertise, but no robot expertise. Platform-independence of code generators and RTS ultimately enables their re-use and further facilitates robot task development with LightRocks.

We also have developed a combined graphical and textual editor for convenient modeling of assembly tasks by factory floor workers, domain experts, and robotics experts. The editor provides two views: one for modeling and one for model execution. It further allows parallel textual and graphical modeling, parsing, and well-formedness checking of tasks and their constituents. The editor is built with MontiCore's text editor generation features, hence the text editor itself is generated: a corresponding editor grammar allows to define keywords, outline elements, filters, and other features from which text editor plugins for Eclipse[2] are generated. The graphical editor is also implemented as an Eclipse plugin and uses the Standard Widget Toolkit[3] (SWT) to render tasks, skills, and actions.

Figure 7 shows the editor's execution view with the textual editor top middle and the graphical editor bottom middle. The graphical editor displays the currently edited network of tasks parallel to the corresponding textual model. Model parsing, context condition checks, outline and syntax highlighting of the text editor are directly re-used from the LightRocks toolchain. The model execution framework takes care of monitoring and representing the currently executed parts of the model. The contents of textual and the graphical editor are synchronized directly, by either informing the textual or the graphical editor about any modifications of the model. Once the developer starts modeling, the graphical editor invokes MontiCore to either parse the changed textual model or prints the changed model into the displayed text editor. Layout data are stored separately and do not pollute the textual model.

The right part of the editor shows the three assembly process levels and their execution states at run-time. The editor highlights the currently executed action and its parents. The top section shows the process level and highlights the active task, the middle section shows the task level and highlights the active skill, and the bottom section shows the skill level and highlights the active action. Currently, the editor does not support on-line editing of LightRocks models at run-time. While desirable, we yet need to ascertain the requirements on valid run-time changes and the implications on error handling mechanisms.

V. Case Studies

We have evaluated LightRocks with KUKA LBR robots and Lego Mindstorms robots. With the former, we modeled

[1]FreeMarker template engine: `http://www.freemarker.org`

[2]Eclipse project: `http://www.eclipse.org/`

[3]SWT website: `http://www.eclipse.org/swt/`

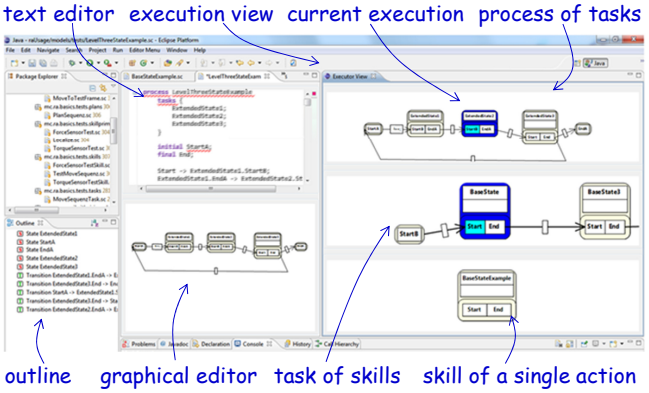

Fig. 7. The model execution view shows the currently executed action and its parents parallel to graphical and textual editors.

classical assembly tasks. Due to hardware restrictions of the Lego robots, we examined whether modeling of non-assembly tasks is feasible with LightRocks. To our satisfaction, modeling of re-usable logistics tasks with these robots was straightforward as well. The following sections briefly report on both case studies.

A. KUKA LBR Assembly Tasks

First case studies were performed as typical assembly tasks with a KUKA LBR manipulator. These modeled tasks included screwing, picking, stacking, plugging, and different kinds of movements (e.g., force controlled). The domain model for this case study comprised of 13 interfaces for various concepts of domain and robot. Figure 8 shows the LBR successfully stacking blocks as modeled with LightRocks. The process modeled to grasp and stack a tower of four blocks consists of one task, which in turn consists of four skills of between one and six actions.

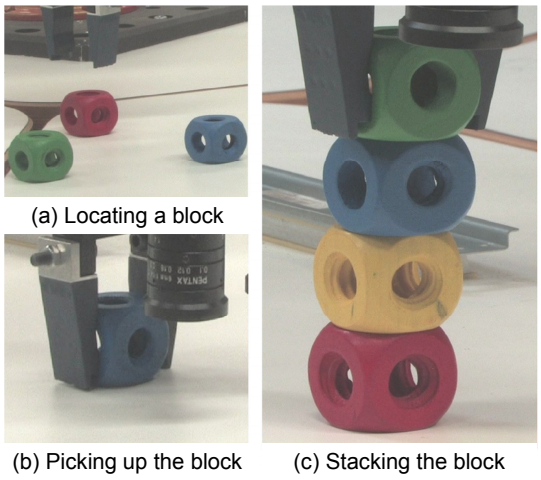

Fig. 8. A LBR robot stacking colored blocks.

Another typical assembly process is plugging workpieces onto another. We therefore modeled a task for the LBR to plug safety sockets on a top-hat rail [1]. Figure 9 shows the performance of the LBR. The executed process model consists of a single task that is repeated once per safety socket. The task itself consists of five skills of up to five actions. Modeling assembly processes for the LBR was straightforward and we could re-use tasks, skills, and actions intuitively. We also observed that most re-use took place on skill level, where - from a human perspective - simple behavior was composed from actions. Skills regarding movement and grasping were re-used most often.

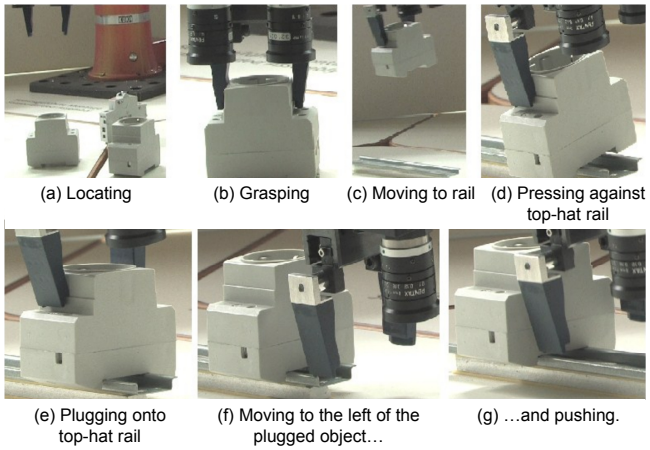

Fig. 9. A LBR plugging safety sockets on a top-hat rail.

B. Lego NXT Logistics Tasks

We also deployed LightRocks to Lego Mindstorms NXT robots to evaluate its usage in different use cases. The Lego robots are designed for education and easy access to robotics. Consequently, their hardware is restricted: out of the box, there are neither laser scanners, nor compliant manipulators. As LightRocks is not tied to platforms providing such hardware, we designed a clean up scenario and modeled the processes accordingly. In this scenario, a robot explores a fixed area while searching for colored blocks. Whenever a block is detected, it is gripped and collected in a container. The robot consists of a base with four wheels and a manipulator (Fig. 10). The base uses a light sensor to ensure moving within the defined area's boundaries, a front-mounted distance sensor to detect blocks, and a manipulator to collect blocks. The manipulator uses a color sensor to detect the blocks' colors.

The LightRocks process to clean up colored blocks consists of three tasks, six skills, and 14 actions. Figure 11 depicts the structure of the process `CollectBlueObjects`, which contains a task `LookForObjects`, and a skill `DriveToNextObject`. Skills and actions interface robot hardware via the leJOS Java operating system[4] as robot API.

Due to lack of memory on the Mindstorms robots, reusing the code generator used with the LBR robot was not feasible: the code generated for the LBR produced too many artifacts for the Mindstorms robot's memory to hold. Instead, we developed a new code generator for the same RTS. Due to the modularity of LightRocks, (a) integrating code

[4]leJOS NXJ website: `http://www.lejos.org/`

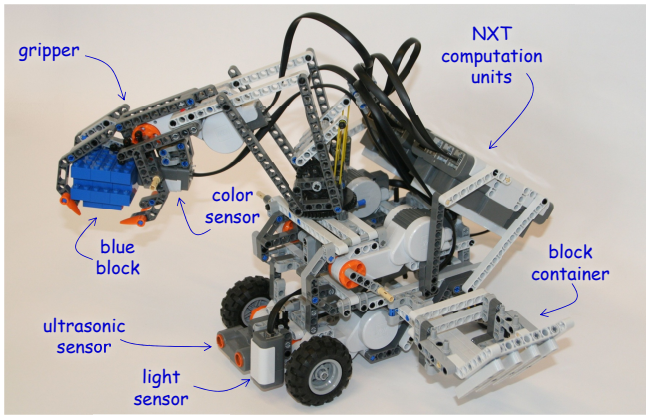

Fig. 10. The Lego Mindstorms robot to perform clean up tasks.

generators is straightforward and (b) the transformation from LightRocks models into SCs is independent of subsequent code generation. Therefore, the new code generator only translates SCs to Java. However, the current Mindstorms version EV3 provides enough memory to re-use the same code generator as used with the LBR.

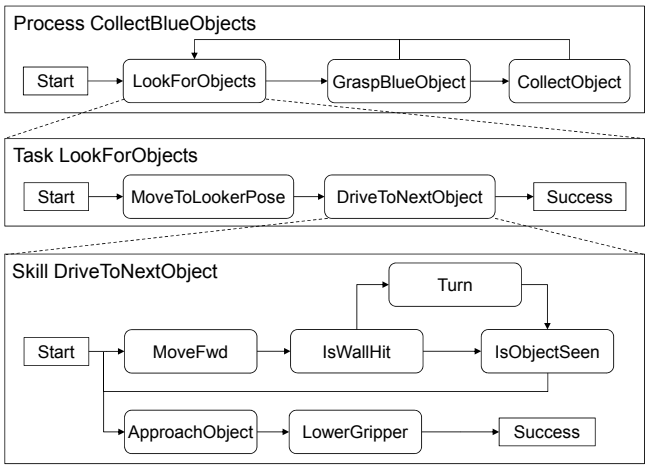

Fig. 11. Process `CollectBlueObjects` with contained task `LookForObjects` and skill `DriveToNextObject`.

C. KUKA iiwa Assembly Tasks

We also applied LightRocks to assembly tasks with a KUKA iiwa robot in a case study with 10 participants. The 3 female and 7 male participants were between 20 and 59 years old and had different degrees of expertise with model-driven engineering, robot programming, LightRocks, tablet computer usage and the iiwa robot. For instance, 60% of the participants had "no" previous experience with the iiwa robot and 20% had "little" previous experience with it.

The participants were given a task and had to answer a questionnaire afterwards. To fulfill the task, the participants had to pick up a lightbulb from its initial position, move it to a thread, screw it into the thread, and activate it via a switch. Figure 12 shows the initial setup with both lightbulb positions. To achieve this, the participants were introduced to the concepts of LightRocks, the iiwa, and tablet UI before they started modeling the task's solutions. The introduction took between 10 minutes and 1 hour, depending on the participant's previous knowledge.

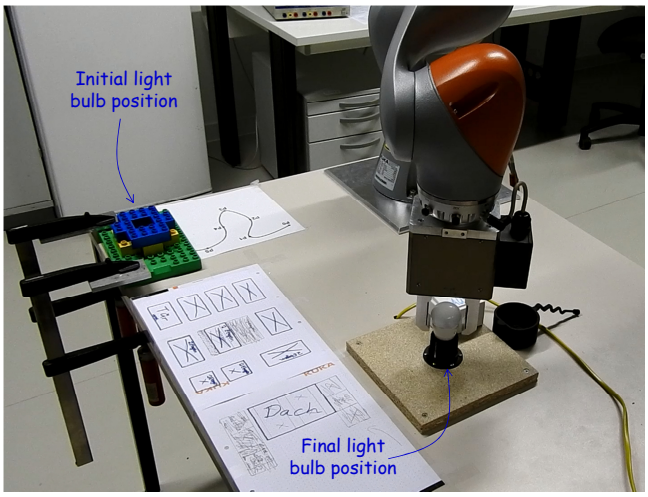

Fig. 12. The case study setup with the lightbulb's initial position on top-left and the target thread bottom-right.

To model the required process and tasks, the participants used a graphical editor displayed in Figure 13 on a tablet computer. In this setup, all skills and actions were provided to the participants, thus no robot API knowledge was required. This corresponds with the idea, that the robot expert provides skills and tasks, and the factory floor worker combines these only. In the end, all participants completed the task. The fastest participant required 45 minutes to complete the complete case study, which included comprehending task description and available skills, teaching relevant poses to the robot, and solving the task. The slowest participant required 2 hours.

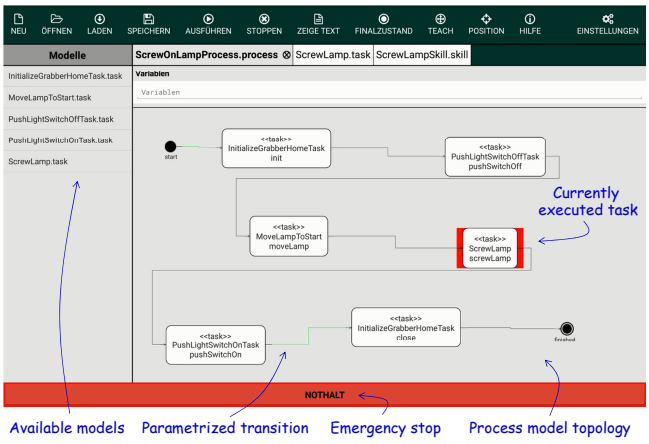

Fig. 13. The tablet-based editor used to model the process and tasks to pickup, deliver, and screw the lightbulb.

This case study was focused on applying LightRocks instead of developing low level skills or actions for it and reflected its usage as intended with factory floor workers.

The study however is biased as 40% of the participants had at least mediocre or good programming skills. As expected, these participants finished faster than the others. In consequence, a future case study will work with participants with little or no programming knowledge only. Nonetheless, even participants without programming knowledge were confident as their feedback included that LightRocks allowed "easy robot programming" and even "enabled untrained users to use robots as tools".

VI. Discussion and Conclusion

LightRocks is a high-level robot programming toolchain feasible for both domain experts and robot experts. Due to the abstraction of LightRocks, processes and tasks can easily be re-used with different robots. Skills and actions are tied to specific platforms but can easily be re-used for different assembly processes. If the new platform represents the same kind of robot (e.g. a LBR with seven degrees of freedom), only the parametrization of the used actions needs to be reconfigured at model level. The LightRocks toolchain furthermore enables re-use of code generators and run-time systems with compatible robots and supports development and execution of assembly tasks with powerful editors. Code generators can be re-used as long as target specific technical restrictions, like available memory or target language are fulfilled. The adaption of provided robot API has to be performed per robot / API version used. Referring to Figure 6 the skill library provider and the code generator developer need to perform target-specific adaption, while the run-time system developer and the application modeler can focus on a target-independent development.

Case studies indicate that LightRocks helps to improve development of robotic (assembly) processes. Nonetheless, the case studies pointed out issues: currently, neither synchronous execution of tasks and skills, nor parametrization of tasks with skills are supported. We will address the issue of synchronous execution of tasks and skills and examine whether such parametrization is useful without raising additional complexities. Actions currently reference a single method of the robot's interface. While facilitating re-use, this also leads to an increased number of actions. In the future, we also will perform further case studies with different robots and differently skilled users.

While the combined graphical and textual editor is helpful at modeling tasks, it currently uses SWT to render tasks. Unfortunately, we have experienced performance issues for large (more than 40 nodes) assembly processes. We therefore will switch to a new rendering engine and we will also examine whether on-line modeling as mentioned in Sect. IV can be realized. Reasoning with tasks and skills will be examined as well. With the strong formalism of preconditions and postconditions, reasoning about the next action to be executed seems useful to assist factory floor workers in modeling robot tasks. Therefore we will examine how to integrate LightRocks with a model of the environment, which, together with the actions, serves as the knowledge base for assembly process reasoning. With these models, the robot may react more dynamically to events and thus increase the flexibility of assembly processes.

References

[1] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A New Skill Based Robot Programming Language Using UML/P Statecharts," in *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, 2013.

[2] J. D. Schutter and J. V. Brussel, "Compliant Robot Motion I. A Formalism for Specifying Compliant Motion Tasks," *The International Journal of Robotics Research*, vol. 7, no. 4, pp. 3–17, Aug. 1988.

[3] T. Hasegawa, T. Suehiro, and K. Takase, "A model-based manipulation system with skill-based execution," *Robotics and Automation, IEEE Transactions on*, vol. 8, no. 5, pp. 535 –544, Oct 1992.

[4] U. Thomas, M. Barrenscheen, and F. M. Wahl, " Efficient Assembly Sequence Planning Using Stereographical Projections of C-Space Obstacles," in *IEEE International Symposium on Assembly and Task Planning*, 2003, pp. 96–102.

[5] U. Thomas, B. Finkemeyer, T. Kroeger, and F. M. Wahl, "Error-tolerant execution of complex robot tasks based on skill primitives," in *IEEE International Conference on Robotics and Automation*, 2003, pp. 3069–3075.

[6] T. Kroeger, B. Finkemeyer, U. Thomas, and F. M. Wahl, "Compliant motion programming: The task frame formalism revisited," in *IEEE International Conference on Mechatronics and Robotics*, 2004, pp. 1029–1034.

[7] J. Maass, S. Molkenstruck, U. Thomas, J. Hesselbach, and F. M. Wahl, "Definition and execution of a generic assembly programming paradigm," *Emerald Assembly Automation Journal*, 2008.

[8] U. Thomas, *Automatisierte Programmierung von Robotern für Montageaufgaben (in German)*. Braunschweig, Germany: Shaker Verlag, 2008.

[9] B. Rumpe, *Modellierung mit UML*, 2nd ed., ser. Xpert.press. Springer Berlin, September 2011.

[10] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel, "MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen," Software Systems Engineering Institute, Braunschweig University of Technology, Tech. Rep. Informatik-Bericht 2006-04, 2006.

[11] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: a framework for compositional development of domain specific languages," in *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, 2010, pp. 353 – 372.

[12] Object Management Group, "OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05)," May 2010, http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[13] M. Look, A. Navarro Perez, J. O. Ringert, B. Rumpe, and A. Wortmann, "Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems," in *Proceedings of the 1st Workshop on the Globalization of Modeling Languages (GEMOC)*, Miami, Florida, USA, 2013.

[14] M. Schindler, *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*, ser. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[15] H. Mühe, A. Angerer, A. Hoffmann, and W. Reif, "On reverse-engineering the KUKA Robot Language," in *First International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, 2010.

[16] J.-C. Baillie, A. Demaille, Q. Hocquet, and M. Nottale, "Events! (Reactivity in urbiscript)," in *First International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, Oct. 2010.

[17] A. Angerer, R. Smirra, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "A Graphical Language for Real-Time Critical Robot Commands," in *Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012)*, 2012.

[18] Hochgeschwender, Nico and Schneider, Sven and Voos, Holger, and Kraetzschmar, Gerhard K., "Towards a Robot Perception Specification Language," in *Fourth International Workshop on Domain-Specific Languages and Models for ROBotic Systems*, November 2013.

[19] C. Schlegel, A. Steck, and A. Lotz, "Model-Driven Software Development in Robotics : Communication Patterns as Key for a Robotics Component Model," in *Introduction to Modern Robotics*, D. Chugo and S. Yokota, Eds. iConcept Press, 2011.

[20] Klotzbücher, M and Smits, Ruben and Bruyninckx, Herman and De Schutter, Joris, "Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, September 2011, pp. 4684–4689.

[21] J. O. Ringert, B. Rumpe, and A. Wortmann, "From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems," in *Software Engineering 2013 Workshopband*, ser. LNI, Stefan Wagner and Horst Lichter, Ed., vol. 215. GI, Köllen Druck+Verlag GmbH, Bonn, 2013, pp. 155–170.

[22] J. Baumgartl, T. Buchmann, and D. Henrich, "Towards easy robot programming: Using dsls, code generators and software product lines," *8th International Conference on Software Paradigm Trends (ICSOFT-PT'13)*, 2013, keywords: Model-Driven Development; DSL; Code generation; Robot; Easy Programming; Software Product Lines.

[23] Vanthienen, Dominick and Klotzbuecher, Markus and De˜Laet, Tinne and De˜Schutter, Joris and Bruyninckx, Herman, "Rapid application development of constrained-based task modelling and execution using Domain Specific Languages," in *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Tokyo, Japan: IROS2013, 2013, pp. 1860–1866.

[24] C. Fitzgerald, "Developing Baxter," in *2013 IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, April 2013, pp. 1–6.

[25] H. Krahn, B. Rumpe, and S. Völkel, "Integrated Definition of Abstract and Concrete Syntax for Textual Languages," in *Proceedings of Models 2007*, 2007, pp. 286–300.

[26] S. Völkel, *Kompositionale Entwicklung domänenspezifischer Sprachen*, ser. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011.

[27] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel, "Textbased Modeling," in *4th International Workshop on Software Language Engineering*, 2007.

[28] D. S. Wile, "Supporting the DSL Spectrum," *Computing and Information Technology*, vol. 4, pp. 263–287, 2001.

[29] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering 2007 at ICSE.*, 2007, pp. 37–54.

[30] H. Krahn, B. Rumpe, and S. Völkel, "Efficient Editor Generation for Compositional DSLs in Eclipse," in *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling 2007*, 2007.

[31] ——, "Roles in Software Development using Domain Specific Modelling Languages," in *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006*. Finland: University of Jyväskylä, 2006, pp. 150–158.