Electronic version of an article published as regular paper in Encyclopedia with Semantic Computing and Robotic Intelligence, Vol. 02, No. 01, 1850005 (2018)
© 2018 World Scientific Publishing Company, https://www.worldscientific.com/worldscinet/escri

https://doi.org/10.1142/S2529737618500053

ENCYCLOPEDIA WITH SEMANTIC COMPUTING
Vol. 2, No. 1 (2018) 1 (?? pages)
© The Authors

Integrating Planning and Reactive Behavior by using Semantically Annotated Robot Tasks

Andreas Schierl† and Alwin Hoffmann† and Ludwig Nägele† and Wolfgang Reif†
†*Institute for Software & Systems Engineering, University of Augsburg, Universitätsstr. 6a, 86159 Augsburg, Germany*
†{schierl, hoffmann, naegele, reif}@isse.de

Received Day Month Year; Revised Day Month Year; Accepted Day Month Year; Published Day Month Year

Tasks that change the physical state of a robot and its environment take a considerable amount of time to execute. However, many robot applications spend the execution time waiting, although the following tasks might require time to prepare. This paper proposes to amend robot tasks with a semantic description of their expected outcomes, which allows planning and preparing successive tasks based on this information. The suggested approach allows sequential and parallel composition of tasks, as well as reactive behavior modeled as state machines. The paper describes the means of modeling and executing these tasks, details different possibilities of planning in state machine tasks, and evaluates the benefits achievable using the approach on two example scenarios.

Keywords: planning; reactive behavior; state machine; robot programming; optimization

1. Introduction

Most classical industrial robot arms are currently used for fixed and recurring preprogrammed tasks. There, planning (at run time) and reaction to indeterministic events are of minor importance. When it comes to more modern or mobile robots, the situation is different: Due to the greater variability in the environment, these robots must be able to sense external events and react accordingly, and can thus be seen as reactive systems.

In this context, it can be observed that when executing a task, performing its physical actions often requires a considerable amount of time compared to the computations needed to prepare them. Still, it is often possible to tell details about the positive outcome of the task even before the task is fully executed. In contrast, for failures the amount of information that can be given in advance is limited (e.g. where the robot will be when the error occurs is usually unknown). While these details could considerably help to plan ahead, many software frameworks often ignore this potential and spend the execution time waiting for completion (cf. Sect. 2).

To improve this situation, we propose to model tasks including their expected outcome(s) and details about the corresponding situation(s). These expected outcomes include successful execution, but also detectable errors and relevant situations that occur during execution. Using these semantically enriched tasks, it becomes possible to plan the following tasks even before the execution of the current task has completed. These tasks can be combined in a sequential or parallel way, can perform case distinctions and can even be used to model reactive behavior through state machines[1]. Composing these tasks, the programmer can configure details about the relationship between the tasks and the quality of transition: For some events, an immediate reaction may be required, so the corresponding reaction task has to be planned before the event can occur, while other events allow some time, so that more relaxed planning schemes can be used.

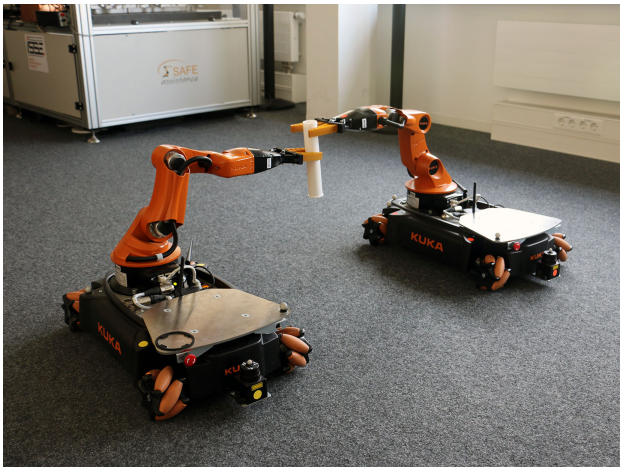

Fig. 1. Robots handing over a baton

To evaluate the approach, the cooperation between two mobile robots was modeled and implemented (cf. Fig. 1). There, two KUKA youBots[2] drive in parallel, and hand over a baton while in motion. To realize the reactive behavior and coordination, state machines are used, while the possibility of preparation during execution reduces the waiting time between the different robot actions. Additionally, a theoretical example with more complex planning is introduced and analyzed, showing the possible advantages of the appraoch.

The work presented here is based on the conference paper on predictive preparation of state machine tasks[3] and extends it mainly in three fields:

This is an Open Access article published by World Scientific Publishing Company. It is distributed under the terms of the Creative Commons Attribution 3.0 (CC-BY) License. Further distribution of this work is permitted, provided the original work is properly cited.

- Semantic descriptions of task outcomes, describing environment states as well as situations that result from changes therein
- Tasks with multiple possible outcomes and their handling
- Complex task compositions, and the advantages and drawbacks of using state machine tasks with deferred preparation

This paper is structured as follows: After an overview of different ways to handle complex robot behavior (Sect. 2), the appraoch of defining and executing tasks is outlined in Sect. 3. Sect. 4 describes the experiments conducted to evaluate the approach and points out the corresponding results. Finally, Sect. 5 gives a conclusion and outlook.

2. Related Work

For modeling and composing tasks or device capabilies, different approaches exist in modern robot frameworks such as *ROS* and *OROCOS*. *ROS* actionlib[4] allows to model interruptible tasks through *Actions*. *Actions* represent long-running, preemptible tasks that provide feedback and notify about their result, and that can be canceled when the goal has to be changed. They can be seen as an extensible way to model device capabilities, allowing extensions by introducing new components that provide *Action* servers. To combine multiple *Actions*, a new component can provide an *Action* that invokes the corresponding *Actions* in parallel or sequentially. However, no timing guarantees are given for this type of composition, as it relies on network communication between the different *Action* servers. Furthermore, *Actions* do not share a common interface and thus cannot directly be passed between different components (for example, a planner cannot provide its result as a generic *Action* that can be passed to another component deciding when it should be executed).

For reactive behavior on the task-level with *ROS*, Bohren et al. introduced *SMACH* state machines[5]. In *SMACH*, *ROS Actions*, *Services* or Python code can be defined as *State*s, and *State*s can be composed into new *State*s in a concurrent, sequential or state-machine form. Therefore, each *State* defines different outcomes that can be used in composite *State*s to define reactions or handle error conditions. Furthermore, states in *SMACH* (and thus state machines) can be made available to other components as *actionlib Actions*. This composition mechanism is similar to the one presented in this paper, however without timing guarantees, so that guaranteed reactions to events cannot be specified this way. Furthermore, *SMACH* (as well as *actionlib*) provides no explicit semantics or metadata for *Action* or *State* post conditions, making further planning during execution harder to achieve. *SMACH* state machines are executed in a blocking way and thus do not offer the possiblity of using the execution time of robot actions for further (motion) planning steps.

For *OROCOS*, restricted Finite State Machines (*rFSM*)[6] allow modeling reactive behavior. An *rFSM* describes a hierarchical state machine without parallelism, aimed at the coordination of robot applications. According to the *pure-coordination* pattern[6], these state machines only process and raise Boolean events which have to be provided by monitor components or handled by configurator components, which in turn manipulate or reconfigure the active components in order to achieve the goal. *rFSM* state machines are implemented on top of the programming language *LUA* with specialized memory allocation and garbage collection, and can thus be executed with real-time guarantees.

As an extension to *rFSM* state machines, Scioni et al. describe how to achieve *preview coordination*[7]: In this approach, the execution environment takes hints about execution probabilities based on likelihood labels on some transitions. Using these labels, likely successor states can be prepared (performing some of their work) while the previous state is still active, as long as there is no conflict between the preparation steps and the actions performed in the current state. This allows reducing the execution time, while keeping the action definitions coherent (instead of moving the preparation step into the previous state).

The preview coordination mechanism introduces a form of decoupling between workflow and capability execution and makes use of meta data about conflicts between states, but the *rFSM* mechanism still does not include further semantic descriptions for the results of states. This way, the following tasks cannot analyze and prepare for the expected results of the previous *State*, a powerful and important feature offered by the approach introduced in this paper.

Another related approach has been introduced by Angerer[8] in the form of Robotics API *Activities*. It can be seen as the basis of the approach suggested in this paper, but has a stronger focus on real-time and specific ways of modeling meta data, and does not support reactive behavior in the form of state machines. This approach has been further extended by Schierl[9] and along with further research led to the results presented in this paper.

3. Approach

To facilitate planning during execution, we propose to model individual robot tasks (cf. Sect. 3.1) along with a semantic description of their expected outcome(s) as described in Sect. 3.2. Based on these outcome description(s), it becomes possible to prepare the successor task (or at least start planning) before the current task has been fully executed – maybe even before execution starts. This feature can be exploited during execution, as described in Sect. 3.3. When it comes to more complex tasks with different viable outcomes, an implementation that is based on simple control flow tends to become confusing, so it becomes helpful to be able to compose multiple simple tasks (cf. Sect. 3.4). As an alternative, the desired behavior can be defined in a model-based way, e.g. using the formalism of state machines[1], as described in Sect. 3.5.

3.1. *Task Definition*

Initially, a task to be executed by a specific actuator (e.g. move to a given position, pick up an item, or even bring me a beer) is modeled as an *Activity* (cf. Fig. 2). As one task can be performed more than once (if you are with friends), it becomes helpful to keep track of different task executions. So, when a task is to be executed in a given situation, its *Activity* is instantiated through *createHandle()* yielding an *ActivityHandle*.

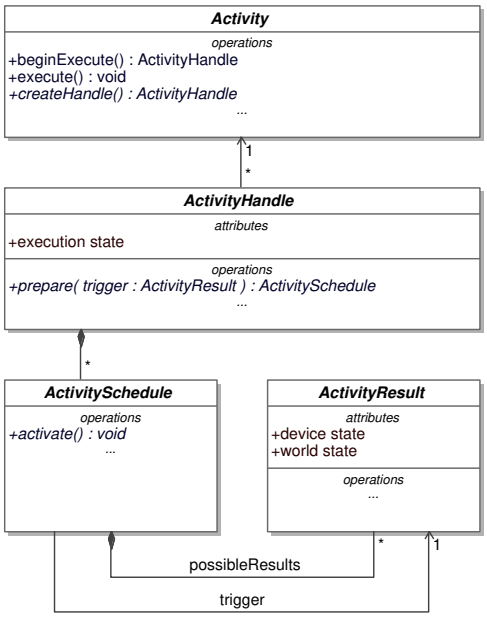

Fig. 2. Relevant classes for task definition and execution

Being responsible for a single task execution, the *ActivityHandle* offers state tracking that notifies about execution progress, and is also responsible to decide how the desired task is to be executed in a given situation. It receives a semantic description of the situation (see below) the task is to be executed in, and has to provide instructions on how to act, and what outcome to expect after the action has been performed. Thus, the *ActivityHandle* takes a situation description called *ActivityResult*, and provides an *ActivitySchedule* that describes the task to execute, as well as a set of possible *ActivityResult*s that can occur while/after executing the task. This way, it provides possible post conditions for the execution of the task, while the preconditions are derived from the results of the previous task. If the previous task has multiple possible outcomes, the *ActivityHandle* is provided with the different acceptable outcomes, and can plan for each of the envisioned situations. This differs from typical planning mechanisms that are based on pre/post conditions and try to combine fitting actions, but instead allows applying a given task in many different situations.

To describe the expected situation, the *ActivityResult* contains status information about each individual actuator contributing to the task, as well as a description about the envisioned geometric and logical situation of the environment, i.e. the world state. The actuator-specific information may contain details about a gripper, e.g. whether it is open or closed, as well as joint positions, velocities and accelerations for robot arms.

3.2. *Task Outcomes*

To talk about the world state, a model as described by Schierl et al.[10, 11] is used. It is based on physical objects and geometric features (modeled as frames), along with relations that are defined between the frames and objects and form an undirected multigraph. Each relation describes an aspect about the relationship between two objects (such as physical objects or coordinate frames), and can give give quantitative information. Between two physical objects, a containment relation tells that a screw is part of a work piece, or a link is part of a robot. Other relations bring together geometric features (modeled as coordinate frames) and their physical object, while relations between coordinate frames describe (quantitative or qualitative) geometric aspects. Between two frames, a logical relation describes the type of connection (e.g. persistent for the connection between two links of a robot, or transient for the connection between a gripper and the grasped work piece), while a geometric relation describes the exact position where the frames are relative to each other (or gives a sensor or computation that provides that information). Together with containment and geometric feature relations, this semantic modeling allows to derive that a gripper is connected to the robot, that the robot has grasped a work piece and that the work piece is at a specific position in space.

While these relations are known by their corresponding objects, they are not static during run time (an object grasped by a gripper can be placed on the ground, removing one relation and establishing up another). Thus, situations with different (geometric) relations have to be modeled. One situation of established relations is modeled as a *FrameTopology*. A *FrameTopology* stores the existing relations for each of the objects modeled in the environment, and allows to find all relations for a given object, as well as relation sequences between given objects.

To give a description about the expected world state after the execution of an *Activity*, the view of the world model is expressed as a *FrameTopology*. When a robot picks up an object from the ground, the relation between ground and object is removed, and a new relation between gripper and object is established. These topology changes are relevant for further planning, because a grasped object changes the shape of a robot: some motions that were previously possible now result in collisions. Thus, the world view modeled in the *FrameTopology* provided by the task includes sets of removed and added relations.

Apart from the established relations, a *FrameTopology* can also give hints about the expected sensor values for geo-

metric aspects to express the positions. This allows to model not only that the robot has grasped an object, but also the position where the robot is expected to be. For tasks that include robot motions, the resulting *FrameTopology* thus contains a position change that gives the (expected) new pose of the robot or all of its components. This geometric information is used whenever performing geometric calculations based on the world view, e.g. when planning the next motion.

To model different situations, the *FrameTopology* does not contain a full snapshot of all relations, but rather works by storing the difference compared to another topology. This is especially important when trying to apply the topology changes after successful task execution: If each *FrameTopology* contained the complete set of relations, it would be impossible to execute two tasks at the same time, because the second task would overwrite the changes applied by the first task, because these changes would not be present in the snapshot taken before the execution of the first task. Instead, modeling *FrameTopology*s as a difference to a previous world state allows to keep all unrelated parts of the situation unchanged. In contrast, the position changes stored in the *FrameTopology* do not have to be applied, because they only describe expected sensor values for the corresponding time. Thus, after successfully executing the task the position of the robot has changed as defined, so that the real sensor values for the robot position are similar to the values predicted by the task.

3.3. *Task Execution*

Once a task has been defined, i. e. created as an *Activity*, it can be executed in a blocking or non-blocking way. For a specific execution instance, the *Activity* creates an *ActivityHandle* that takes the *ActivityResult*s of the previous Activity and prepares *ActivitySchedule*s for each of the situations. The *ActivitySchedule*s are activated so that they trigger the execution of the respective task once their *ActivityResult* is reached. Additionally, the *ActivitySchedule*s provide their possible *ActivityResult*s for use with following *Activity*s.

The first option is to execute the task in a blocking way. Fig. 3 shows the tasks (A) and (B), which are executed sequentially in a blocking way. In this case, the method *execute()* of task (A) blocks until a completion *ActivityResult* of a corresponding *ActivitySchedule* has been reached.

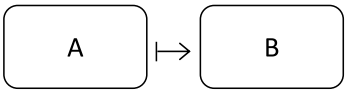

Fig. 3. Blocking execution of task (A), followed by (B)

Fig. 4 shows the timing of this behavior. The *Activity*s are shown as horizontal life lines with time running from left to right, while the different boxes denote the phases preparation *P* and execution *E*. However, in this case the situation description contained in the *ActivityResult* is of little use, because once the result has been reached, the described situation is already reached, and the execution time has already been wasted waiting.

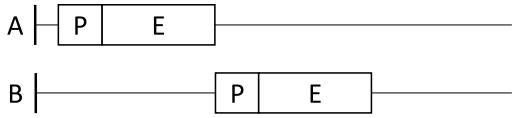

Fig. 4. Blocking execution of task (A), followed by (B), with preparation (P) and execution (E)

To use the execution time, non-blocking execution can be used. Fig. 5 shows task (A) followed by task (B) in a non-blocking way. Executing tasks with *beginExecute()* allows the control flow to continue once any of the *ActivitySchedule*s has been triggered and the execution of the task has thus started. The following task can then be prepared for different possible start situations (based on the *ActivityResult*s of the currently running task), and immediately execute one of the planned solutions if the corresponding situation occurs.

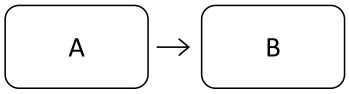

Fig. 5. Non-blocking execution of task (A), followed by task (B)

This situation is shown in Fig. 6 – here, the dotted lines denote times where (B) is already planned and waiting to be executed. The preparation of (B) happens while (A) is running, and (B) starts running once (A) is completed.

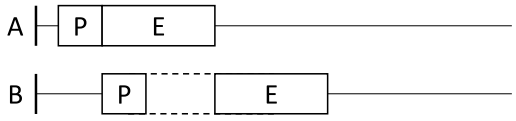

Fig. 6. Non-blocking execution of task (A), followed by task (B), with preparation (P) and execution (E)

In contrast, if task (B) is independent of (A), e.g. if it controls different devices, it starts immediately when prepared, as shown in Fig. 7. However, this execution mode does not guarantee parallelism, because preparation of (B) starts after execution of (A) has started, resulting in a certain delay between the two execution starts.

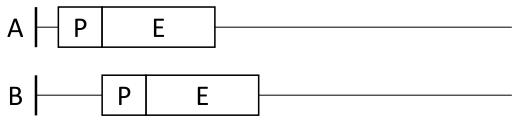

Fig. 7. Timing of non-blocking execution of independent tasks, with preparation (P) and execution (E)

While called non-blocking, this execution scheme is not fully asynchronous, because preparation for the following task only starts after the current task has started. This behavior becomes obvious when having a look at a sequence of three tasks (as shown in Fig. 8), each of which has two possible results. In this situation, preparation of (B) starts once (A) has started, while (C) is prepared once (B) has started.

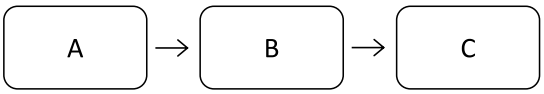

Fig. 8. Non-blocking execution of multiple tasks

Using the proposed non-blocking semantics (as shown in Fig. 9), (B) has to be prepared for both results of (A), yielding two *ActivitySchedule*s. However, since preparation of (C) is postponed to the time when (one concrete schedule for) (B) is started, (C) only has to be planned for the chosen (B) schedule and its two results.

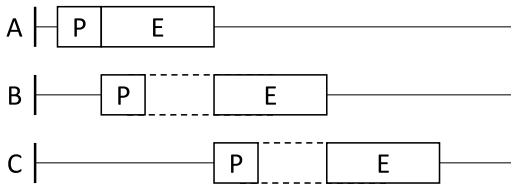

Fig. 9. Timing of non-blocking execution of multiple tasks, with preparation (P) and execution (E)

In a fully asynchronous execution, in contrast, preparation had to be performed for both *ActivityResult*s of each *ActivitySchedule*s of (B), resulting in four *ActivityResult*s to plan for and an exponential growth in preparation work for longer sequences.

3.4. *Task Composition*

In addition to different execution types, multiple tasks can be combined to provide better guarantees about execution (e.g. reliable parallel or sequential execution).

When sequentially combining two tasks (as shown in Fig. 10), the created sequential task provides the second task with all the *ActivityResult*s of the first task's *ActivitySchedule*s, so that the second task can react to all expected outcomes of the first task, while the results of the second task are provided as results of the seqence. In cases where determinism is required, this kind of preparation can even allow real-time guarantees (given a corresponding software framework that supports scheduling of real-time tasks, e.g. the Robot Control Core introduced by Vistein et al.[12]): preparing the second task for all possible outcomes of the first task before the first task's execution starts guarantees that the second task can immediately take over no matter how long or short the first task takes.

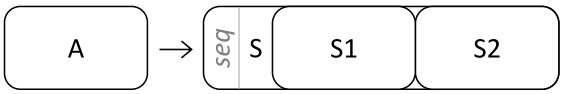

Fig. 10. Simple task, followed by a sequential task

In Fig. 11, the sequence *S* is executed after (A), and the preparation of *S1* and *S2* completely happens before *S1* is started, guaranteeing that *S2* can be started immediately when *S1* ends. However, this determinism is only possible if the outcomes are fully known in advance: if details of an outcome description depend on sensor data, these *ActivityResult*s can only be provided when the sensor data becomes available, and preparation of the next task is delayed to this moment (no longer guaranteeing determinism).

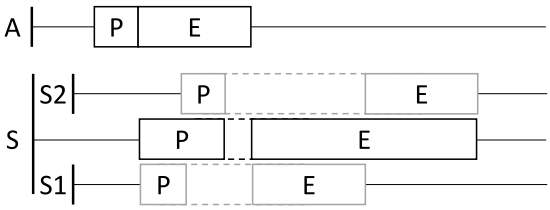

Fig. 11. Timing of sequential task with preparation (P) and execution (E)

However, these sequences do not delay preparation, so the second task has to be planned for all results of the first task, which has to be planned for all results of the previous task, leading to an exponential amount of preparation required for longer sequences.

As a second combination option, parallel execution can be used (as shown in Fig. 12). The created parallel task then provides both subtasks with the same initial situation (as *ActivityResult*s), and combines resulting *ActivitySchedule*s (and optionally *ActivityResult*s) to create the *ActivitySchedule* and *ActivityResult*s of the parallel task.

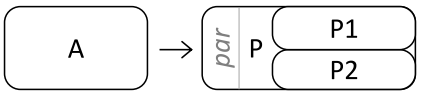

Fig. 12. Simple task, followed by a parallel task

This situation is shown in Fig. 13, where the parallel tasks *P1* and *P2* are planned and thus allow *P* to start them at the same time after (A) is completed. Of course, the parallel task has to make sure that the subtasks do not conflict, e.g. by checking that they control different devices.

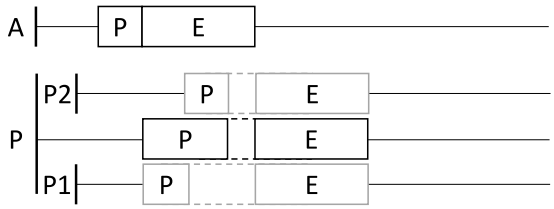

Fig. 13. Timing of parallel task with preparation (P) and execution (E)

Additionally, simple case distinctions can be made based on the possible *ActivityResult*s, as shown in Fig. 14. A task with case distinction classifies the *ActivityResult*s of the previous task and forwards them to the corresponding subtask depending on the decision condition.

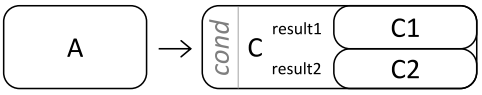

Fig. 14. Simple task, followed by a conditional task

Fig. 15 shows the execution of this situation, with (C) as a case distinction for the result of (A), deciding to execute (C2) and unload (C1).

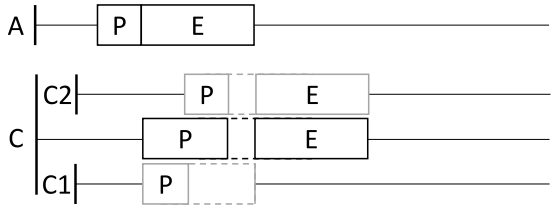

Fig. 15. Timing of parallel task with preparation (P) and execution (E)

Combined tasks can further be composed to handle more complex scenarios. Fig. 16 shows a task *S* that first executes (A1), and continues with (C1) in case of success, or with (B1) followed by (C1) if (A1) fails.

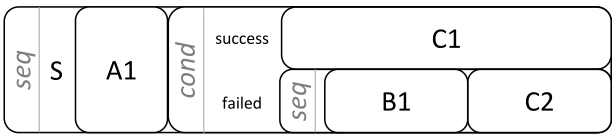

Fig. 16. Complex task with sequential and conditional composition

For execution, first (A1) is prepared. For its error result, (B1) has to decide how to react, while (C1) handles all results of (B1). Additionally, (C2) has to be prepared for the success case of (A1). Once all these preparations are completed, execution of *S* can begin, first executing (A1), and then switching to (B1) (because (A1) failed) and (C1). This behavior is shown in Fig. 17.

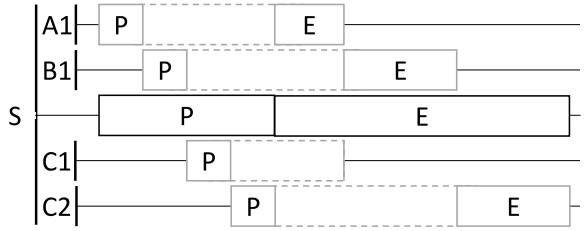

Fig. 17. Timing of complex task execution with preparation (P) and execution (E)

This entire composition can provide real-time guarantees for execution, because every possible case has been prepared for beforehand; however as a downside all possible cases have to be planned, which requires a lot of preparation time before execution can start. Additionally, this type of composition does not allow for unbounded loops, because preparation requires to unroll the loop to plan for every possible amount of repetitions, and quickly limits bounded loops by the exponential amount of results to plan for.

Instead, these repetitions should be handled through non-blocking execution of separate tasks that are repeated through control flow mechanisms wherever timing guarantees are less important, or through state machines that allow to better handle reactive behavior as described in the following section.

3.5. *State Machines for Reactive Behavior*

For longer or repeating task sequences, these composition mechanisms that fully plan ahead may reach their limit, because the number of tasks that has to be prepared grows exponentially with the number of successive tasks (given that each has more than one possible result). Thus, it becomes helpful to limit the amount of preparation. One way to achieve this while supporting complex compositions of tasks is the use of state machines – a step often taken when specifying reactive behavior.

The proposed approach uses *Activity*s to define states of the state machine. For each state (or *Activity*), transitions can be given that specify a switch to another *Activity* if certain *ActivityResult*s occur. Additionally, one distinct *Activity* is chosen as start state, and *ActivityResult*s of some *Activity*s can be defined as transitions to a final state.

As each state can be entered more than once during execution, for its *Activity* multiple *ActivityHandle*s, *ActivitySchedule*s and also *ActivityResult*s are prepared. The resulting set of *ActivityResult*s for an *Activity* is dynamic, so the transition cannot give a complete list. Instead, it works as a predicate to *ActivityResult*s that chooses whether a given *ActivityResult* qualifies for the transition, which can be as generic as *any error* or as specific as *any successful execution of the given grasp*.

As an example, Fig. 18 shows a state machine of the three tasks (A), (B), and (C), switching from (A) to (C) if (A) succeeds, and to (B) if it fails. State (B) is always followed

by (C), leading to two different paths reaching (C).

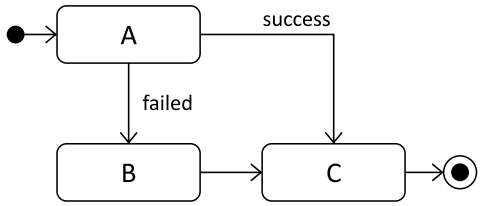

Fig. 18. Definition of a state machine task consisting of three tasks

Regarding execution, state machine tasks differ from the composed tasks mentioned previously: While for sequential, parallel or conditional tasks preparation can completely be performed up front, state machine tasks require some preparation while the task is running. The *Activity* for the initial state is prepared traditionally, based on the *ActivityResult*s of the previous task. Then, the *ActivityResult*s of the created *ActivitySchedule* are compared against the transitions originating from the start state to decide whether they have to be handled by switching to another state (i.e. *Activity*). However, for these transitions and *Activity*s, different options exist when to prepare them for the corresponding results, as a trade-off between preparation of unnecessary traces and not having prepared a required task. For the scope of this paper, the preparation is usually delayed until their originating state is entered (i.e. its *Activity* started running), preparing the target states for outgoing transitions using the reported *ActivityResult*s of the current state, so that they can react to the occurrence of the corresponding result. Additionally, the outgoing transitions of the target state are analyzed, so that they can be handled once the target state is entered. This way, the states are prepared with a look-ahead of one transition, which works fine as long as the typical execution time of a state is longer than the preparation times for all following states.

Fig. 19 shows the preparation timing resulting for the state machine, numbering the different traces leading to state (C) as (C1) and (C2). When executing the state machine, (B1) and (C1) are prepared once (A1) is started, and when the failure occurs, (B1) is started and (C1) is unloaded. A new instance (C2) has to be prepared for the new situation resulting of (B1), which is then executed once (B1) finishes. Using this method yields a behavior that is similar to the one of non-blocking execution (cf. Fig. 6), where the following task is prepared once the previous is started.

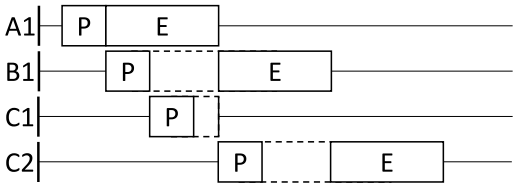

Fig. 19. Execution of state machine from Fig. 18 with preparation (P) and execution (E)

However, this mechanism of delaying preparation outgoing transitions once the state is entered is not sufficient to guarantee that a certain transition can be taken without further delay, especially if the result happens quickly after entering the state. Thus, for critical transitions, preparation may not be delayed until this time.

To allow this, transitions can also be annotated with qualifiers that direct the order or timing of preparation. For transitions that may not be missed (such as error recovery strategies that bring the robot into a safe state), a transition can be marked as reliable, requesting it to be prepared before the originating *Activity* starts. These reliable transitions are handled similar to sequential tasks described before, making sure that the following state can be executed no matter how long or short the first state takes. While having the same advantages of sequential tasks, reliable transitions also inherit their drawbacks, especially the amount of preparation necessary if more than one *ActivityResult* is handled by a reliable transition, as well as the fact that cycles in the reliable transition graph are forbidden.

Fig. 20 amends the transition from (A) to (B) with a stereotype *«reliable»*, defining that the execution environment has to guarantee that the transition will be taken immediately if the failure occurs.

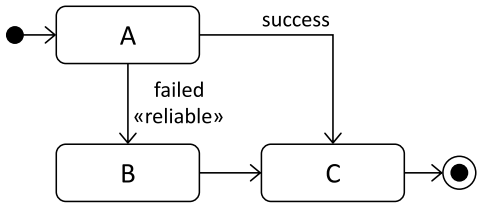

Fig. 20. Definition of a state machine task with a reliable transition

For execution, reliable transitions and their target state have to be prepared along with their originating state, before the originating state is entered. This behavior is shown in Fig. 21. Here, (A1) and (B1) are planned before (A1) is started, making sure that the occurrence of the failure cannot happen while (B1) is still being planned.

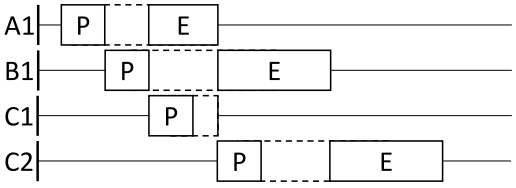

Fig. 21. Execution of reliable transitions from Fig. 20 with preparation (P) and execution (E)

When all transitions in the state machine are marked with the stereotype *«reliable»*, the defined task has the same semantics as the complex composition shown in Fig. 16, guaranteeing that every transition can be taken immediately. For

execution, all traces through this state machine have to be prepared beforehand, so preparation looks like the one shown for the complex composition in Fig. 17. This leads to long preparation time before execution can start, and inherits the same restrictions that long reliable sequences (with multiple results per task) lead to exponential growth in preparation time, and reliable cycles are forbidden.

4. Experimental Results

The proposed approach has been evaluated in two examples. First as a real-world example, the interaction between two mobile robots has been modeled. Two mobile robots drive parallel to each other and hand over a baton while in motion. The implementation uses two KUKA youBots and is based on the Robotics API[8]. There, both robots are controlled using their on-board computers using a C++ control core[12], while the high-level coordination and task execution is implemented in Java and performed from a laptop computer connected to the youBots through a wireless network.

Because the handover example (apart from error handling) is purely sequential, the first implementation was based on separate tasks for moving the arm and gripper and the execution model described in Sect. 3.3. For all tasks that do not depend on the second robot, non-blocking execution was used, while the correct order of the gripping and releasing tasks was performed through blocking execution of the corresponding tasks in a common control flow. While working fine in simulation, this implementation led to unintended delays between the grasp and release operations when executed on real robots, and thus required more space for parallel driving than necessary. This was mainly caused by the unreliable network connection and inefficient network code that required multiple communication roundtrips to transmit and start the tasks.

A second implementation modeled the expected behavior through two linked state machines (cf. Fig. 22). Here, the following tasks could already be prepared while the second youBot was waiting for the first youBot, so the delay between the grasp and release operation as well as the required space for the interaction were significantly reduced.

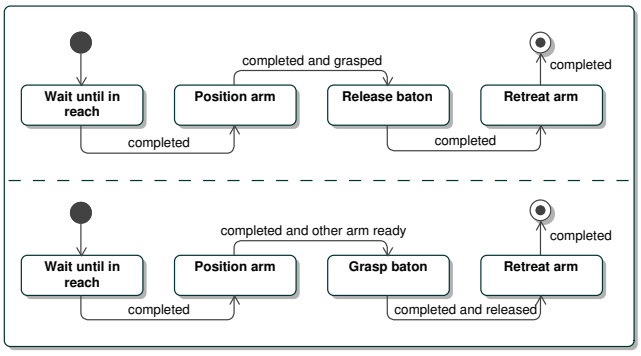

Fig. 22. State machine: Robot interaction

The second, theoretical scenario was performed without connection to a robot framework, but based on a prototypical implementation with support for reliable transitions. Here, the scenario was to drive a mobile robot to a pick-up area, take a large object, and return to its start position. In addition, during all steps a sensor had to be observed, and the robot had to stop once the sensor detected a dangerous situation. To complicate things, in the environment the shortest path to the pick up zone could not be taken while carrying the large object, but instead a detour was required.

Fig. 23 shows a state machine model of the task. The top row of states gives the main success flow, while reliable transitions handle danger occuring during this task. Additionally, a *Back off* state has been added, which is executed after the robot has stopped in case of emergency. As after stopping the robot is already in a safe state, this transition is not time-critical and thus does not need to be reliable.

The motion planning tasks, *Go to pick up zone* and *Return home* were implemented as tasks that take considerable time to plan (3 s for collision free motion planning) and to execute (5 s to move the robot along the planned path), while *Pick up* only takes time to execute (3 s to move the arm and gripper), but plans quickly (1 s). As a comparative reference simulating a system that does not support planning ahead based on result meta data, the three main success task were executed in a blocking way as a sequence.

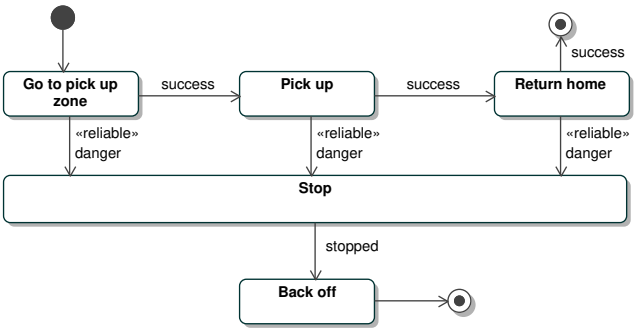

Fig. 23. State machine: Second scenario

When executed, the state machine of the second scenario implementation completed after about an average of 16.05 s, while the sequential execution took an average of 20.08 seconds. Fig. 25 shows the resulting life lines for those different execution models, clearly showing where time can be saved by preparation during execution.

5. Conclusion

Working with robots, the performance of applications is not only limited by the available processing power of the computer, but also by the physical limitations of controlled devices. Usually, the time needed to execute a task is significantly longer than the mere computation time, but still some

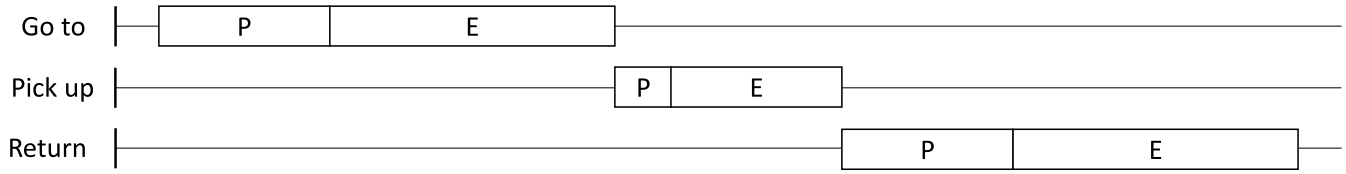

Fig. 24. Sequential execution of the second scenario, with preparation (P) and execution (E)

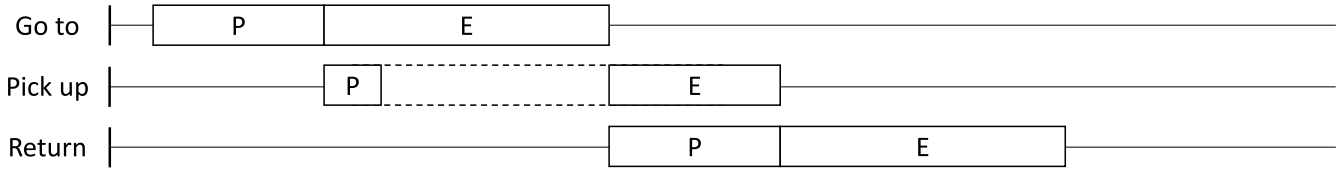

Fig. 25. State machine execution of the second scenario, with preparation (P) and execution (E)

planning also takes considerable time. In this paper, we proposed to amend robot tasks with descriptions of their expected outcomes. This allows preparing the following task while the current task is still running, thus reducing delays between the execution of successive robot tasks and avoiding the situation that execution time goes unused although preparation steps may be pending.

These amended tasks can be executed in a blocking or non-blocking way, or combined into complex tasks. Besides sequential and parallel composition, state machines promise to allow the specification of more complex reactive behavior including recurring subtasks, while still behaving like regular tasks, so that further composition or preparation remains possible.

A prototypical implementation of this mechanism has been created based on the Robotics API[8] and shown to work for cooperating robots. Additionally, the approach promises to accelerate the execution of tasks where considerable time is spent preparing the next steps, such as collision free motion planning, and also allows specifying events for which a timely reaction has to be guaranteed (given a capable execution environment, such as a Realtime Robot Control Core[12] used with the Robotics API).

Still, this paper is limited to simple preparation strategies of state machines (transitions can be guaranteed, or planned when the state is entered). As a part of further research, more complex strategies might benefit from likelihood annotations[7] or estimated preparation times to decide which transitions in the state machine will likely happen and should be prepared first (to reduce the risk of missed transitions). Additionally, longer sequences of transitions could be prepared in corresponding situations, e.g. to skip over states that take a very short time to plan and execute. In the given theoretical example, this could help if the grasping task took shorter, because then the execution time of the *Go to* task could be used to plan the *Return* task.

References

[1]D. Harel, Statecharts: A visual formalism for complex systems, *Sci. Comput. Program.* **8**, 231 (June 1987).

[2]R. Bischoff, U. Huggenberger and E. Prassler, *KUKA youBot - a mobile manipulator for research and education*, *Proc. 2011 IEEE International Conference on Robotics & Automation, Shanghai, China*, (IEEE. pp. 1–4.

[3]A. Schierl, A. Hoffmann, L. Nägele and W. Reif, *Integrating reactive behavior and planning: Optimizing execution time through predictive preparation of state machine tasks*, *2018 Second IEEE International Conference on Robotic Computing (IRC)*, (2018).

[4]E. Fernandez, E. Marder-Eppstein and V. Pradeep, *actionlib* Online, accessed Feb 2016, `http://wiki.ros.org/actionlib`.

[5]J. Bohren and S. Cousins, The SMACH high-level executive, *IEEE Robotics & Automation Magazine* **17**, 18 (2010).

[6]M. Klotzbücher and H. Bruyninckx, Coordinating robotic tasks and systems with rFSM statecharts, *Journal of Software Engineering for Robotics* **3**, 28 (2012).

[7]E. Scioni, M. Klotzbuecher, T. De Laet, H. Bruyninckx and M. Bonfe, *Preview coordination: An enhanced execution model for online scheduling of mobile manipulation tasks*, *Proc. 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2013)*, (Nov 2013). pp. 5779–5786.

[8]A. Angerer, A. Hoffmann, A. Schierl, M. Vistein and W. Reif, Robotics API: Object-Oriented Software Development for Industrial Robots, *Journal of Software Engineering for Robotics* **4**, 1 (2013).

[9]A. Schierl, Object-oriented modeling and coordination of mobile robots, PhD thesis, Universität Augsburg, (2017).

[10]A. Schierl, A. Angerer, A. Hoffmann and W. Reif, *Consistent world models for cooperating robots: Separating logical relationships, sensor interpretation and estimation*, *2017 First IEEE International Conference on Robotic Computing (IRC)*, (April 2017). pp. 101–108.

[11]A. Schierl, A. Hoffman and W. Reif, Consistent geometric estimation based on a world model describing logical relationships and sensor interpretation, *Journal of Software Engineering for Robotics* **8**, 104 (2017).

[12]M. Vistein, A. Angerer, A. Hoffmann, A. Schierl and W. Reif, Flexible and continuous execution of real-time critical robotic tasks, *International Journal of Mechatronics and Automation* **4**, 27 (1 2014).