2016 IEEE International Conference on Robotics and Automation (ICRA)
Stockholm, Sweden, May 16-21, 2016

Human-Robot Collaborative High-Level Control with Application to Rescue Robotics

Philipp Schillinger*, Stefan Kohlbrecher†, and Oskar von Stryk†

***Abstract*— Motivated by the DARPA Robotics Challenge (DRC), the application of operator assisted (semi-)autonomous robots with highly complex locomotion and manipulation abilities is considered for solving complex tasks in potentially unknown and unstructured environments. Because of the limited a priori knowledge about the state of the environment and tasks needed to achieve a complex mission, a sufficiently complete a priori design of high level robot behaviors is not possible. Most of the situational knowledge required for such behavior design is gathered only during runtime and needs to be interpreted by a human operator. However, current behavior control approaches only allow for very limited adaptation at runtime and no flexible operator interaction. In this paper an approach for definition and execution of complex robot behaviors based on hierarchical state machines is presented, allowing to flexibly change the structure of behaviors on the fly during runtime through assistance of a remote operator. The efficiency of the proposed approach is demonstrated and evaluated not only in an example scenario, but also by application in two robot competitions.**

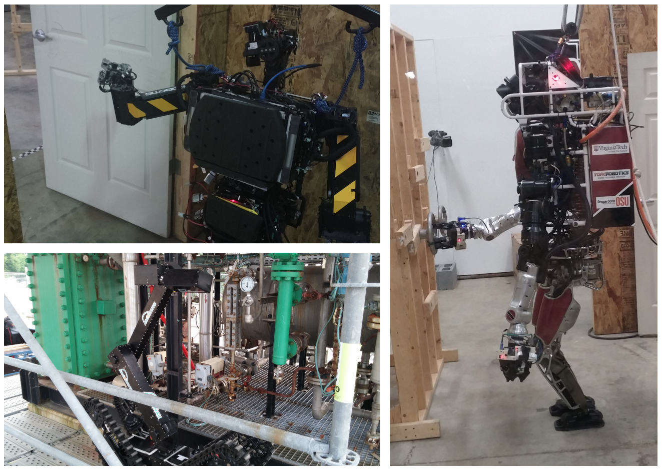


Fig. 1. Some of the robots controlled by *FlexBE*[2], the ROS implementation of the presented approach.

I. INTRODUCTION

Recent research demonstrates significant progress in individual robot capabilities such as navigation, manipulation, or perception. However, the focus is often very narrow and limited to a particular, well defined scenario and only considers specific capabilities instead of a comprehensive scenario. Approaches in the area of high-level control typically build upon these well defined capabilities. Although some research has been done towards robust control in the presence of disturbances, a general understanding of the task and the environment is usually assumed.

Furthermore, a robot is typically either meant to be controlled by a human operator and completely relying on this operator for high-level decisions, or designed to act autonomously without considering human commands during runtime. Being able to combine both approaches would enable a robot to efficiently operate even in scenarios which have not been completely or correctly modeled in advance. Unfortunately, very few approaches allowing for a flexible collaboration between a human operator and a mostly autonomous robot exist, enabling to adapt to unforeseen difficulties during runtime.

In order to address this gap, a high-level specification is required which not only is able to coordinate executed actions without operator involvement, but also provides a way to let the operator adapt the robot to gained situational knowledge whenever the operator decides this is required. The ability of arbitrary adaptation is a crucial requirement for versatile rescue robots to be useful under real-world conditions. Similar challenges are encountered in space and deep-sea robotics and thus also these applications can benefit from the proposed approach.

The approach presented in this paper provides an efficient solution for human robot collaboration and introduces several mechanisms to model the ways of interaction between both of them, flexibly ranging from directed teleoperation to full autonomy. Special attention has been paid to scenarios where a previously specified robot behavior is not sufficient to solve the desired task or might even cause the robot to fail without considering situational knowledge gained during runtime. In order to address these scenarios, our approach enables to incorporate such observations. Since this level of runtime adaptation might introduce new sources of failure, as discussed in Section III, our approach assists the operator in making these adaptations and performing consistency checks in order to increase robustness.

Our approach uses ROS and has been evaluated in real world scenarios and competitions, demonstrating the ability of adapting to unfolding situations and reacting to failures. The most prominent application has been in the DARPA Robotics Challenge (DRC), for which our approach was originally designed. Two of the teams used the developed behavior engine during the DRC finals.

* Robert Bosch GmbH, Corporate Research, Department for Cognitive Systems, 70442 Stuttgart, Germany philipp.schillinger @de.bosch.com (presented work has been conducted while at †)

† Department of Computer Science, Technische Universität Darmstadt, 64289 Darmstadt, Germany {kohlbrecher, stryk}@sim.tu-darmstadt.de

[2]Available online as open-source software: http://flexbe.github.io

II. RELATED WORK

In recent years, several behavior modeling approaches have been shown to reliably coordinate available actions in order to solve pre-defined tasks.

The behavior-based approach, as presented in [1], relies on the definition of several simple, independent behaviors which are then combined to specify the high-level behavior of the system. As stated by [2], a characteristic of these systems is their reactive nature, making them less applicable for complex scenarios. [3] targets this problem by using reinforcement learning, [4] and [5] apply fuzzy logic. However, since the mission specifications are not exactly known in advance for our application, it is not possible to sufficiently train the robot using such an approach. Also, a human operator is hard to model in this context.

The behavior control framework of [6] coordinates between simple behaviors in a similar way as behavior-based approaches, but single behaviors can be modeled explicitly by state machines using a state controller library. SMACH [7] implements the concept of hierarchical state machines with each state represented by a single *Python* class. XABSL [8] uses XML for specification and has been extended in [9] to use a custom programming language which is converted to XML when compiled. [10] uses XML as a intermediate language as well, generated from a GUI.

While these approaches focus on specifying control strategies for autonomous execution, recent work also considers some sort of situational adaptation and provides basic ways for collaboration with an operator based on pre-defined interaction schemes.

In order to assist the creation of shared autonomy approaches, [11] examines human robot interaction and proposes a taxonomy for selecting an appropriate level of autonomy. [12] presents a framework for robot navigation tasks which supports different modes corresponding to different levels of autonomy. Furthermore, [13] extends the considerations of [12] and investigates the effects of temporal latency in shared autonomy systems. Coactive Design [14] finally shifts the focus away from pure robot autonomy towards interdependence in collaboration. As investigated in [15], this approach leverages the close coupling in joint activities of humans and robots.

In the DRC Trials, some teams used short scripts for automating motion execution [16], but most competitors relied on operators for selecting actions [17]. In the DRC Finals participants also used task-specific autonomous behaviors. Their use was favored by prior knowledge about tasks being available, reducing the need for online adaptation compared to application in a real disaster [18].

In summary, none of the available approaches allows to specify complex high-level control strategies while supporting arbitrary modifications to them during execution. Thus, main contribution of this paper is providing a high-level control engine which is runtime-adaptive to unforeseen situations, but allows autonomous operation while the specifications match the assumed scenario [19].

III. MAIN CHALLENGES

Given the scenario of an operator wanting to modify running code during a rescue mission or similar critical tasks, there are several risks that have to be considered. Therefore, this section lists the main challenges which come along with identified risks and discusses provisions to be taken in order to address them.

A. Distraction

One serious problem introduced with providing the capability to perform runtime modifications is distraction of the operator. When concentrating on making changes to a running behavior, the operator of the robot may miss critical events occurring to the robot and thus may fail to handle them correctly. However, distraction can be avoided by choosing a multi-operator approach similar to [20]. A primary operator is dedicated to ensuring safe task execution and ready to intervene whenever required. An additional behavior supervisor solely monitors behavior execution and is able to perform modifications if needed.

When splitting the workload between multiple operators, communication is a critical factor. Especially for performing runtime modifications, the operator responsible needs a good understanding of both when to conduct changes and what to change.

B. Time Constraints

Even rather small changes to a behavior require some time and significant modifications even more so. During this time, the robot would continue executing the current behavior which can be seen twofold. On the one hand, when editing robot behavior, it is helpful that the robot does not interrupt the execution of the current task, especially when the changes have trivial effects. On the other hand, critical adaptations, intended to prevent task failure, likely require the robot to wait for these changes and stop execution, at least at a point in time before failure may occur.

This yields the requirement of being able to pause behavior execution when making changes. Pausing execution can also ensure consistency across behavior versions, which is discussed in Section V. Nonetheless, it is important to preserve the interruption-free character of trivial modifications.

C. Runtime Failure

The most important challenge, however, is guaranteeing successful execution of specified behaviors. This concern increases even more in case of runtime modifications. The changed behaviors are not only prone to containing new possible sources of errors due to their creation under operator stress during task execution, but also because they have not been tested in-depth before their application.

In order to address this challenge, several mechanisms need to be added for ensuring both, early avoidance and detection of errors, as well as robustness in case a behavior fails during runtime. A first approach for avoiding errors is relying on model-based behavior specification along with automatic code generation instead of manually written code.

Also, before sending a modified behavior to the robot, consistency checks are performed in order to detect higher-level modeling mistakes and semantic errors. Finally, at the robot, as described in detail in Section V, behavior modifications are initialized independently from the currently executed version and are only applied if this step was successful. When executing a behavior, failure of this behavior leads, in the worst case, to stopping it, but does not affect other components or future behavior execution. A modified, corrected version can be started immediately afterwards.

IV. PRELIMINARIES

This section defines a basis for the high-level control approach presented in this paper. Based on these definitions, the novel concepts provided by our approach are proposed in the next section. As discussed in Section II, state machines are the most common and extensible concept of modeling the high-level behavior of a system. Therefore, our approach extends the basic model of hierarchical state machines. In this context, behavior refers to how the robot agent interacts with its environment in a defined manner and thus corresponds to the specifications of the state machine hierarchy.

Definition: Behavior. *A behavior B specifies how the robot is meant to interact with its environment and is implemented by a corresponding state machine B_{SM}.*

Behaviors rely on basic capabilities of the robot and, based on the current situation, coordinate which action should be executed next. These actions, depending on the robot's capabilities, are modeled as single states and constitute the atomic building blocks of any composed behavior.

Definition: State. *A state $s \in S$ interfaces basic robot actions with high-level behaviors. Each state defines a set of outcomes s_{Oc} and execution of a state is terminated by returning one outcome matching the result of its corresponding action $oc \in s_{Oc}$.*

In other words, a state is considered to be active while a specific action is being executed. Thus, states refer to action execution, rather than action results or the situation the robot currently is in. While the robot is in a certain state, it continuously monitors its performance, warns if unknown events occur, and returns one of its outcomes when the respective outcome conditions are fulfilled. Modeling states in such a way allows for flexibility since it can be abstracted from how the action changes the environment or the internal configuration of the robot in detail.

Definition: State Machine. *A state machine $SM = (S_{SM}, t_{SM})$ composes a set of states S_{SM} where each state $s^{(i)} \in S_{SM}$ is the instantiation of a state implementation $s \in S$. When the active state returns an outcome $oc^{(i)} \in s^{(i)}_{Oc}$, the corresponding transition function of the state machine $t_{SM} : S_{SM} \times s_{Oc} \rightarrow S_{SM}$ defines which state becomes active next.*

The hierarchical character of a state machine enables to recursively use other state machines as states. Furthermore, by interpreting defined behaviors itself as capabilities of the robot, even they can be used as states and incrementally define more abstract and complex behaviors. Note that in the above definition, a distinction is made between state implementations, S, and state instantiations, S_{SM}, of a specific state machine. This provides a basis for later distinguishing between parts of a high-level specification that can be modified during runtime and others that are fixed.

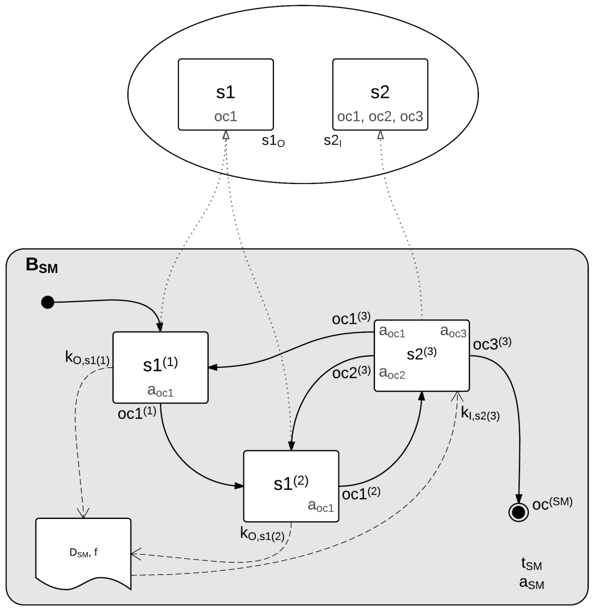


Fig. 2. Summary illustration of the presented concepts. At the top, state implementations form the basic robot capabilities and the schematic behavior state machine at the bottom composes instantiations of these states by connecting their outcomes and associating *userdata* keys.

In addition to the control flow, a state machine also has to coordinate the data flow. This is relevant, as the input data of one action corresponding to a certain state typically depends on the output data of previous actions which can be referred by different states. In order to address this issue, [7] added a concept called *userdata* which we incorporate in this context.

Definition: Userdata. *Each state machine SM additionally defines userdata D_{SM}, represented by a set of key-value pairs where $f : K_{D,SM} \rightarrow V_{D,SM}$ maps each specific key $k \in K_{D,SM}$ to its respective value. States define userdata keys they need, s_I, and provide, s_O.*

Passing data from state A to state B with $s^{(A)}, s^{(B)} \in S_{SM}$ is thus defined as $f(k_{I,sB}) \mid k_{I,sB} = k_{O,sA}$ where $k_{I,sB} \in s^{(B)}_I$ is an input key of $s^{(B)}$ and $k_{O,sA} \in s^{(A)}_O$ is an output key of $s^{(A)}$. For example, a state corresponding to an action which moves an arm of the robot would need data regarding the trajectory how the arm should be moved and could expect a corresponding userdata input. A different state, handling planning of such a trajectory given a target joint configuration or endeffector pose, would provide the required data as an output.

V. OPERATOR INTERACTION

Interaction between the robot in the field and a human operator is a trade-off that needs to be carefully balanced and should be adaptive to meet different situations best. If there is too much interaction, low communication bandwidth and stability will significantly slow down robot performance. Furthermore, given the exceptionally stressful role of an operator in rescue scenarios, a human is prone to make mistakes or forget important details when step-wise telling the robot what to do. On the other hand, less interaction typically means less adaptability to unforeseen events. A human is much more capable to reason about unexpected changes or inaccuracies while a robot would fail to address these correctly or execute a pre-defined strategy.

In order to cope with this trade-off, the presented approach consists of two main concepts. First, an adaptive *Autonomy Level* helps to mark critical decisions within behaviors with several levels of required autonomy in advance. During mission execution, the operator is able to adjust the autonomy of the robot and thus grants it the responsibility to make safe decisions on its own, but ask for approval for more critical decisions. Furthermore, the operator is able to force any decision of the robot, regardless of what how the onboard behavior progresses. Second, runtime modifications of the whole behavior structure enables the operator to not only make small adjustments of what the robot is supposed to do, but also completely re-design the planned approach while the robot already is in the field.

A. Collaborative Autonomy

Based on modeling robot behavior as hierarchical state machines where states correspond to actions and transitions relate to decisions of the robot, an approach of regulating the robot's autonomy can be effectively coupled with state transitions by potentially blocking critical decisions. This prevents the robot from continuing with actions based on possibly incorrect decisions.

Definition: Autonomy Level. *In a state machine SM, each outcome of a state $oc^{(i)} \in s_{Oc}^{(i)}$ defines an autonomy guard $a_{oc}^{(i)}$ which can block the corresponding transition $t_{oc,i} := t_{SM}(s^{(i)}, oc^{(i)})$. During execution, the current autonomy level is determined by a_{SM} and only allows the execution of transitions $t_{oc,i}$ if $a_{oc}^{(i)} < a_{SM}$.*

All transitions which are not allowed to be triggered autonomously are suggested to the operator and require explicit manual confirmation to be executed. This mechanism ensures that transitions are only executed when their required outcome is reliably detectable. Furthermore, the operator does not have to wait for the robot to request a certain outcome, but can trigger one of the state outcomes manually. This already enables an adjustable autonomy ranging from directed teleoperation, by requiring the operator to perform all transitions manually, to full autonomy. This works well as long as the structure of the behavior still matches the situation assumed when it has been defined.

B. Runtime Modifications

Referring to Section III, the approach of making runtime modifications to behaviors has to respect constraints in order to address possible issues. For both improved robustness against runtime failure and efficiency regarding robot communication, changes made to a behavior must have discrete synchronization. Not each atomic change can immediately be visible to the high-level executive, as replacing one state by another, for example, would otherwise leave an inconsistent model between deletion of the old state and creation of the new one. Instead, the operator has to be able to explicitly trigger when to synchronize behavior changes, ensuring a correct and consistent version.

Considering discrete synchronization of changed behaviors, runtime modifications can be expressed as switching from one behavior to a more recent, but almost arbitrarily different version of this behavior, denoted by $B \to B'$.

Along with a switch, the set of state instantiations of the state machine S_{SM} may be changed by adding new states and removing existing ones: $S'_{SM} := S_{SM}^{(+)} \cup S_{SM} \setminus S_{SM}^{(-)}$. Also, the properties of each state instantiation $s^{(i)}$, such as parameter values, may change even if the state itself remains in S'_{SM} and if $s^{(i)}$ is an inner state machine, the states of it can change as well. This refers to the distinction of state implementations from state instantiations as discussed in the previous section. While the set of state instantiations of a state machine can be changed as described, the set of available state implementations of the system remains the same and builds a basis for efficiently modeling the changes.

In addition to the set of states itself, also the connection of its elements is subject to change. This refers to the changes in the control flow, represented by the transition function, as well as changes to the data flow, represented by the *userdata* keys. While re-arranging the execution order of independent states or adding new states in between is relative unproblematic, constraints imposed onto the data flow are much more critical.

Although the concept of *userdata* is a powerful yet simple mechanism for passing data between states, it gets challenging when new keys are added during runtime after some parts of the state machine have already been executed. In principle, there are two sources for the *userdata* keys. First, there is default *userdata* known to the state machine from the beginning. Each entry specifies a default value, so this would be no problem for behavior modification. But second, each output key of each executed state is implicitly added to the set of available *userdata* keys and its value depends on the results of the state execution. In order to ensure safe runtime modifications, all keys being added during a modification should be added to the default *userdata*, holding a default value for when there is a read access before a write access.

VI. IMPORTANT CONSIDERATIONS

Besides defining a concept for how switching high-level specifications during runtime can be modeled, the main challenges as discussed in Section III have to be addressed explicitly for this complex form of robot-operator interaction.

A. Consistency

Maintaining consistency is a critical objective for realizing behavior modifications. The most important part is consistency regarding the current state, which is tightly related to the challenges of time constraints and runtime failure. Let $s_a^{(i)}$ denote the fact that $s^{(i)}$ is the currently active state. In between starting to modify the behavior and executing the switch, the behavior may be invalid. If now, while the operator is modifying the behavior B, $s_a^{(i)}$ triggered one of its outcomes $oc^{(i)}$ and thus caused B_{SM} to transition to the next state $s_a^{(i+1)} := t_{SM}(s^{(i)}, oc^{(i)})$, the currently modified and maybe still invalid version B' would need to be updated, too. Otherwise, the following switch of $B \rightarrow B'$ would also cause a switch of the active state $s_a^{(j)} \rightarrow s_a^{(i)\prime}$ with $j \neq i$. However, updating B' is not trivial as well. Even if B'_{SM} is valid when performing the update, there are constellations in which the update would fail or behave wrongly. For example, if $s_a^{(i+1)} \in S_{SM}^{(-)}$, there is no chance to update B'. But also if t_{SM} changed according $s^{(i)}$ such that $s^{(k)} := t'_{SM}(s^{(i)}, oc^{(i)})$ where $k \neq i+1$, the behavior update would wrongly make $s^{(k)}$ the active state.

These problems are solved by locking the active state. Locking a state means that the actions performed by this state are still executed, thus, it does not require to interrupt desired actions. However, when all actions are finished and the transition function would trigger the next, possibly faulty state, the current state remains active until being unlocked.

Consistency across behavior modifications can be achieved by requiring a lock to be active and only permitting modifications to parts outside the locked state. This also gives the operator enough time for doing changes as discussed in Section III. Nevertheless, locking should preferably be done after the robot detected the problem itself and suggests a failure outcome or in advance if the supervising operator recognizes that future actions might lead to failure.

Keeping in mind that we are dealing with hierarchical state machines, locking a single state may be too restrictive. Inner state machines appear as simple states to their respective parents and hence may also be locked. A locked state machine keeps executing and only if it reaches one of its outcomes, it requires to get unlocked in order to trigger a transition to an outside state.

B. Switching Procedure

The implemented procedure for updating a running behavior is depicted in Figure 3. As determined by the switch conditions, applying runtime modifications is only possible if the running behavior is locked and respects the previously discussed consistency requirements. These checks, as well as initialization of the new version, happen decoupled from the active behavior. This mechanism primarily addresses the challenge of runtime failure as discussed in Section III.

In the realized framework, behaviors are implemented as *Python* classes. Runtime modifications are achieved by patching the executed source code and re-importing this respective part, while keeping the locked part running and injecting it back into the re-imported behavior definition.

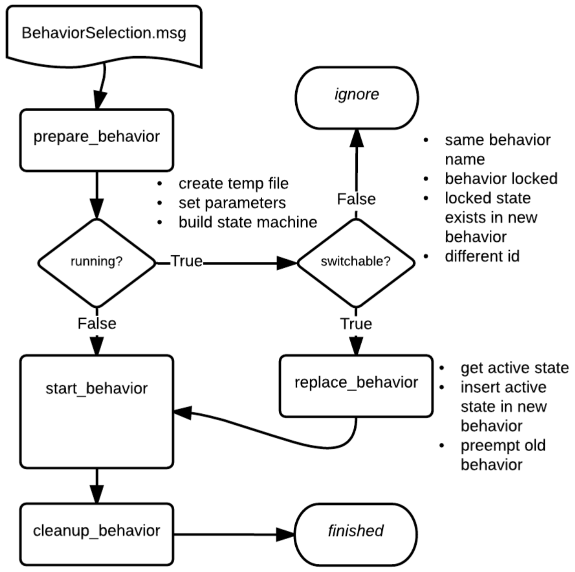


Fig. 3. Procedure of a behavior switch. The current behavior is only interrupted after the new version passed all verification checks.

C. Code Generation

Finally, code generation addresses the challenge of operator distraction and reduces the likelihood of runtime failures. When specifying changes to a behavior, the operator necessarily takes the role of a software developer, equivalent to designing a behavior in advance. In order to reduce the cognitive load, keep the distraction imposed by making these changes low, and enabling operators with no expert knowledge to make adjustments, state machines are intuitively modeled based on available state implementations. From this model, the executable source code defining a behavior is then automatically generated. If more flexibility is required in certain situations, an expert operator is still able to make arbitrary adjustments in the source code afterwards.

VII. EXPERIMENTS

The presented approach has been implemented using ROS and is available as the open-source behavior framework *FlexBE*[3]. It provides an extensive user interface for designing and executing behaviors as well as a runtime-safe onboard executive. As part of the evaluation of the capability to react to unforeseen situations, *FlexBE* has not only been evaluated using a carefully designed scenario, but also integrated into multiple systems and practically applied in two different robot competitions.

A. Example Scenario

To demonstrate the applicability of *FlexBE* in a complex system, we use a simulated scenario[4] based on monitoring an industrial site after an earthquake, as depicted in Figure 4. The robot has to autonomously patrol the site and measure the state of valve levers and dial gauges at four checkpoints in order to decide if some machines need to be shut down.

[3]Source code at: https://github.com/team-vigir/flexbe_behavior_engine
[4]Video recording available at: https://youtu.be/zmsr8f4P5Zk

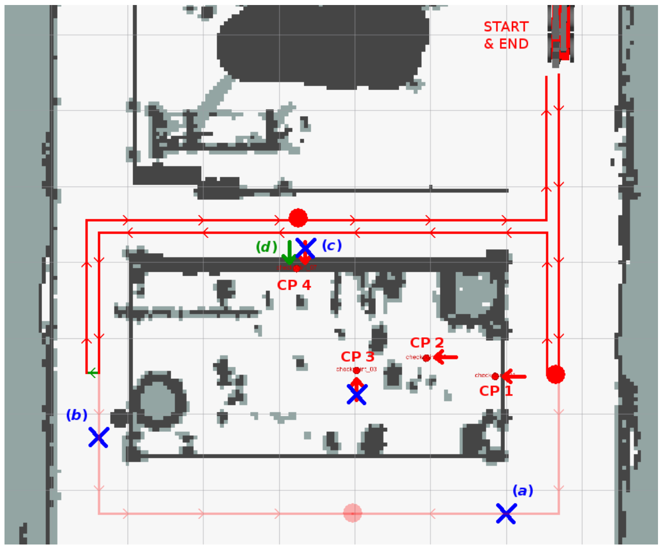


Fig. 4. Example scenario including the planned mission (red), occurred events (blue), and runtime adaptations (green). Crosses indicate problems.

Defining this mission can be done by using the statemachine editor of *FlexBE* based on diverse generic robot capabilities such as navigation, arm positioning, and measuring, provided as parameterizable state implementations to *FlexBE*.

In the simulated scenario, parts of the damaged structure occasionally collapse onto the walkway, block the robot, and eventually damage its sensor arm. This requires the operator to help out and perform runtime adaptations to the executed behavior in order to account for the new circumstances and prevent mission failure. Figure 5 provides an overview of the applied reactions based on the capabilities of *FlexBE*.

The *Collaborative Autonomy* becomes useful after the first unexpected event (*a*) happened. Given that the mission is not running as initially planned, the operator chooses that the robot requires confirmation for some of its critical decisions while autonomously making rather uncritical ones by reducing autonomy to an intermediate level. This helps to prevent task failure at (*b*) and gives the operator a chance to slightly adjust the executed behavior by abolishing checkpoint 3.

Larger *Runtime Modifications* are required as the robot fails to measure checkpoint 4 due to a damaged arm (*c*). It is still able to perform measurements, but unable to lift its arm high enough. The flexibility of *FlexBE* lets the operator redefine this part of the behavior on-the-fly, adding a motion of the tracked flippers of the robot in order to reach the required height. In the initially defined behavior, such a usage of the flippers was not even considered. After applying these modifications (*d*), the robot is able to autonomously finish the mission despite the unexpected difficulties.

	Event	Robot	Operator
a	Path to CP 3 blocked	Autonomously replans path	Precautiously reduces autonomy
b	Alternative path blocked, CP 3 is now unreachable	Suggests mission failure due to unreachable CP 3	Abolishes CP 3, declines failure, mission continues
c	Robot fails to lift arm due to on-site damage	Misses to detect failure, proposes success	Modifies behavior to account for the damaged arm
d	Applied behavior modifications	Continues task autonomously	Nothing to do

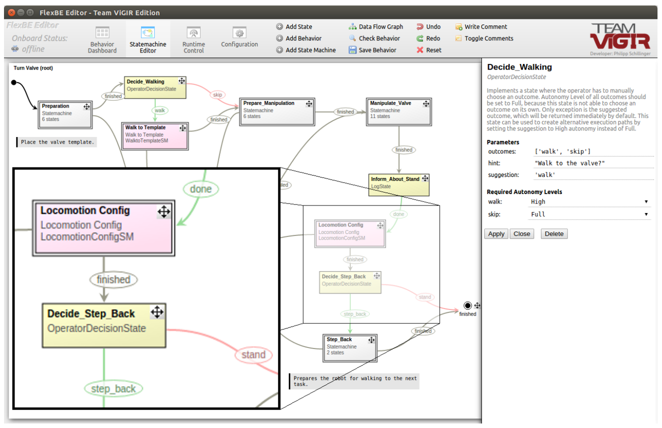
Fig. 5. Important events and reactions in the presented example scenario.

B. *Real-World Application*

FlexBE has successfully been applied in the DARPA Robotics Challenge[5] (DRC) finals and the ARGOS Challenge[6]. Figure 6 shows one of the behaviors executed by *Team ViGIR's* ATLAS robot in preparation for the DRC finals. The modular design paradigms based on the presented concepts have proven to be very useful for efficiently designing complex behaviors. Typical task-level behaviors designed for the DRC Finals or similar tasks contained around 50 to 100 states in their hierarchy, practically intractable by solely text-based definitions.

Especially for behavior execution, *FlexBE's* monitoring and runtime control capabilities facilitated both safe execution in the presence of possible failure and relieving the operator in controlling the robot while everything works as expected. During the earlier tests with the system, the option to modify a behavior at runtime helped *Team ViGIR* a lot for better adapting the task strategies to the previously not in detail known capabilities of the robot and thus to increase the performance of their approaches. In the case of *Team Argonauts*, this runtime adaptation helped even more. Due to an unexpected interruption and partial reset in one of their runs during the first competition of the ARGOS Challenge, the previously designed strategy was not applicable anymore. By quickly adjusting the behavior during their run in the user interface, which just took a couple of seconds and did not involve restarting any component, the robot was again able to carry out the remaining part of the task autonomously. Without such an adaptation, according to the competition rules, this would have meant the end of this run and thus significantly lowered the team's performance.

[5]Further information: http://www.theroboticschallenge.org
[6]Further information: http://www.argos-challenge.com



Place the valve template.

Prepares the robot for walking to the next task.

Decide_Walking
OperatorDecisionState

Implements a state where the operator has to manually choose an outcome. Autonomy Level of all outcomes should be set to Full, because this state is not able to choose an outcome on its own. Only exception is the suggested outcome, which will be returned immediately by default. This state can be used to create alternative execution paths by setting the suggestion to High autonomy instead of Full.

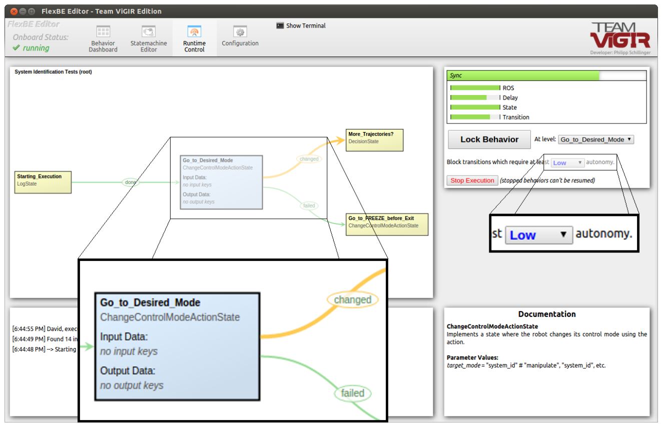
Parameters

outcomes:	['walk', 'skip']
hint:	"Walk to the valve?"
suggestion:	'walk'

Required Autonomy Levels

walk:	High
skip:	Full

Fig. 6. Behavior to solve the DRC task of turning the valve, visualized by FlexBE's statemachine editor. Even during execution, the structure can easily be re-arranged without manually writing code.



[6:44:55 PM] David, exec
[6:44:49 PM] Found 14 in
[6:44:48 PM] --> Starting

ChangeControlModeActionState
Implements a state where the robot changes its control mode using the action.

Parameter Values:
target_mode = "system_id" # "manipulate", "system_id", etc.

Fig. 7. Monitoring behavior execution in FlexBE's GUI. The active state is centered and possible next states are displayed. Since proceeding would require high autonomy in this case, but the *Autonomy Level* is set to "Low", the transition suggested by the robot is highlighted and has to be approved.

VIII. CONCLUSIONS AND FURTHER WORK

In this paper, a novel high-level control approach incorporating operator collaboration has been presented and its main challenges have been addressed. Using our approach, the operator is able to influence mission execution of the robot when required and can even completely re-design the applied high-level strategy. These concepts have been implemented by the open-source behavior framework *FlexBE*.

FlexBE and the presented concepts have been the core part of the high-level approaches of *Team ViGIR* and *Team Hector* in the DARPA Robotics Challenge finals. Furthermore, *FlexBE* has been used by *Team Argonauts* in the ARGOS Challenge to develop autonomous behaviors due to the advanced support for easily designing mission strategies. Although primarily designed for rescue scenarios, *FlexBE* was, for example, also used to implement a social robot entertaining participants at the welcome reception of the *European Conference on Mobile Robots* (ECMR) based on the *STRANDS* EU project. This underlines the broad applicability of the results of this paper.

Further work is currently being conducted regarding the ability of runtime modifications in combination with an approach for behavior synthesis. This enables more sophisticated instructions while still allowing adjusting details of generated behaviors. Furthermore, combining correct-by-construction behavior generation with the flexibility of runtime adjustments provides a high-level interface while guaranteeing safe and reliable execution. Given that a robot is able to detect deviation from an initially defined scenario on its own, it may use this mechanism to autonomously adapt its own behavior specifications on-the-fly.

ACKNOWLEDGMENT

The authors would like to thank David Conner and Spyros Maniatopoulos for helpful support and discussions. Also, thank goes to all members of *Team ViGIR*, *Team Hector*, and *Team Argonauts* for their contributions to the development of the respective robotic systems used in this paper.

This project was partially supported by TOTAL SAS by provision of the industrial scenario within the ARGOS Challenge and the Defense Advanced Research Projects Agency (DARPA) under Air Force Research Lab (AFRL) contract FA8750-12-C-0337.

REFERENCES

[1] M. Matari and F. Michaud, "Behavior-based systems," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer Berlin Heidelberg, 2008, pp. 891–909.
[2] L. De Silva and H. Ekanayake, "Behavior-based robotics and the reactive paradigm a survey," in *Proceedings of the 11th IEEE International Conference on Computer and Information Technology (ICCIT)*, 2008, pp. 36–43.
[3] E. Uchibe, M. Asada, and K. Hosoda, "Behavior coordination for a mobile robot using modular reinforcement learning," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3, 1996, pp. 1329–1336.
[4] S. Kovács, "A flexible fuzzy behaviour-based control structure," *HUCI*, 2003.
[5] R. Huq, G. K. Mann, and R. G. Gosine, "Behavior-modulation technique in mobile robotics using fuzzy discrete event system," *Robotics, IEEE Transactions on*, vol. 22, no. 5, pp. 903–916, 2006.
[6] P. Allgeuer and S. Behnke, "Hierarchical and state-based architectures for robot behavior planning and control," in *Proceedings of 8th Workshop on Humanoid Soccer Robots, IEEE Int. Conf. on Humanoid Robots, Atlanta, USA*, 2013.
[7] J. Bohren and S. Cousins, "The SMACH high-level executive [ros news]," *Robotics & Automation Magazine, IEEE*, vol. 17, no. 4, pp. 18–20, 2010.
[8] M. Lötzsch, "XABSL-a behavior engineering system for autonomous agents," Diploma thesis, Humboldt-Universität zu Berlin, October 2004.
[9] M. Risler, "Behavior control for single and multiple autonomous agents based on hierarchical finite state machines," Ph.D. dissertation, Technische Universität Darmstadt, February 2010.
[10] T. Medina, M. Hybinette, and T. Balch, "Behavior-based code generation for robots and autonomous agents," in *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, 2014, pp. 172–177.
[11] J. M. Beer, A. D. Fisk, and W. A. Rogers, "Toward a framework for levels of robot autonomy in human-robot interaction," *Journal of Human-Robot Interaction*, vol. 3, no. 2, pp. 74–99, 2014.
[12] M. A. Goodrich, D. R. Olsen, J. W. Crandall, and T. J. Palmer, "Experiments in adjustable autonomy," in *Proceedings of IJCAI Workshop on Autonomy, Delegation and Control: Interacting with Intelligent Agents*, 2001, pp. 1624–1629.
[13] K. L. Blatter and M. A. Goodrich, "Modeling temporal latency in the supervisory control of human-robot teams," 2014.
[14] M. Johnson, "Coactive design: Designing support for interdependence in human-robot teamwork," Ph.D. dissertation, TU Delft, Delft University of Technology, 2014.
[15] M. Johnson, J. M. Bradshaw, P. J. Feltovich, C. M. Jonker, B. van Riemsdijk, and M. Sierhuis, "The fundamental principle of coactive design: Interdependence must shape autonomy," in *Coordination, organizations, institutions, and norms in agent systems VI*. Springer, 2011, pp. 172–191.
[16] T. Koolen, J. Smith, G. Thomas, S. Bertrand, J. Carff, N. Mertins, D. Stephen, P. Abeles, J. Englsberger, S. Mccrory, *et al.*, "Summary of Team IHMC's virtual robotics challenge entry," in *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2013.
[17] H. Yanco, A. Norton, W. Ober, D. Shane, A. Skinner, and J. Vice, "Analysis of human-robot interaction at the DARPA Robotics Challenge Trials," *Journal of Field Robotics*, 2014.
[18] DRC-Teams, "What happened at the DARPA Robotics Challenge?" www.cs.cmu.edu/~cga/drc/events, 2015.
[19] P. Schillinger, "An approach for runtime-modifiable behavior control of humanoid rescue robots," Master's thesis, Technische Universität Darmstadt, Department of Computer Science (SIM), 2015.
[20] S. Kohlbrecher, A. Romay, A. Stumpf, A. Gupta, O. von Stryk, F. Bacim, D. A. Bowman, A. Goins, R. Balasubramanian, and D. C. Conner, "Human-robot teaming for rescue missions: Team ViGIR's approach to the 2013 DARPA Robotics Challenge Trials," *Journal of Field Robotics*, 2014.