Design, Execution, and Postmortem Analysis of Prolonged Autonomous Robot Operations

Sebastian G. Brunner, Peter Lehner, Martin J. Schuster, Sebastian Riedel, Rico Belder, Daniel Leidner, Armin Wedler, Michael Beetz, and Freek Stulp

***Abstract*—In the context of space missions and terrestrial applications, both mission goals and task implementations for autonomous robots are becoming increasingly complex. Thus, the challenge of monitoring the achievement of task objectives and checking the correctness of their implementation is becoming more and more difficult. To tackle these problems, we propose an unified architecture that supports different stakeholders during the different phases of the deployment: 1) the design phase; 2) the runtime phase; and 3) the postmortem analysis phase. Furthermore, we implement this architecture by enhancing our task programming framework RMC advanced flow control with powerful logging, debugging, and profiling capabilities. We demonstrate the efficiency of our approach in the context of the ROBEX mission, during which the DLR Lightweight Rover Unit autonomously deployed several seismometers in an unknown rough terrain on Mt. Etna, Sicily. The analysis results for a state machine consisting of more than 1500 states and more than 1900 transitions are presented. Finally, we give a comparison between our framework and related software tools.**

***Index Terms*—Software, middleware and programming environments, field robots, space robotics and automation, autonomous agents, mobile manipulation.**

I. Introduction

ROBOTIC application domains are becoming more and more diverse every day, and now include complex exploration and manipulation tasks in industrial use cases, disaster response missions, and space applications. In the latter context, we deployed a mobile manipulation platform to autonomously setup a network of seismometers on Mt. Etna (see Fig. 1), Sicily, Italy; an environment with many similarities to the moon. This *moon analogue* demonstration mission was conducted with the *Lightweight Rover Unit (LRU)* [1] in the context of the *Robotic Exploration of Extreme Environments (ROBEX)* project. The goal of the mission was to perform seismic measurements with the deployed network in order to infer the geological composition and detailed information about the crust layers of the volcanic target region.

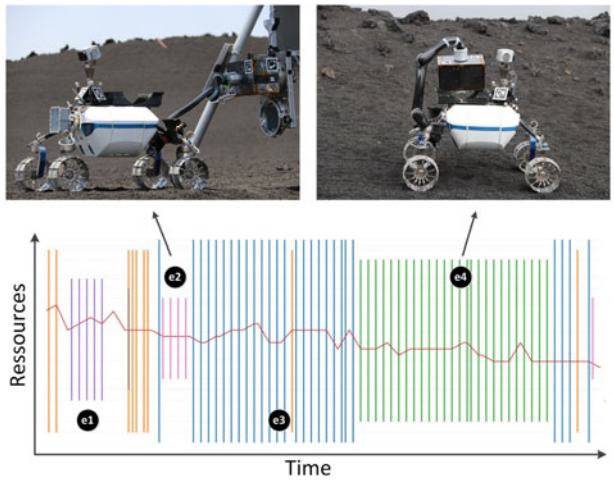

Fig. 1. Photos show two specific events during a mission in the context of the ROBEX project. Left: the LRU rover docks to a seismometer attached to a lander mockup. Right: the rover lifts the seismometer in order to place it onto the ground. The graph below schematically shows a Resource Capability Profile (IRCP-SYA-Bars, see Section III), where the resources a robot is using are tracked. The labels represent critical events during an autonomous task execution.

Apart from the extreme environment (2600 m above sea level, 100 km/h wind speeds), there were two main challenges. First, a high degree of autonomy was required to set-up the seismometers; missions could run for two hours without human intervention, and the mission control center was 30 km away from the demonstration site. Second, many different stakeholders had different interests during the different phases of the mission life-cycle. During the preparation, robot developers were mainly interested in analyzing logged data for debugging purposes. During runtime, operators in the mission control center were interested in monitoring the correctness of the behavior. After the mission, seismologists were interested in the scientific data.

These requirements guided the development of our architecture for the analysis of robotic tasks throughout the whole life-cycle of the robot, i.e. the design, runtime and post-mortem

Manuscript received August 31, 2017; accepted December 28, 2017. Date of publication January 17, 2018; date of current version February 1, 2018. This letter was recommended for publication by Associate Editor A. Agostini and Editor T. Asfour upon evaluation of the reviewers' comments. This work was supported in part by the Helmholtz project alliance ROBEX (HA-304), in part by the European Commission under contract numbers FP7-ICT-608849-EUROC and H2020-ICT-645403-ROBDREAM, and in part by the Collaborative Research Center EASE (SFB 1320) funded by the German Research Foundation (DFG). *(Corresponding author: Sebastian Georg Brunner.)*

S. G. Brunner, P. Lehner, M. J. Schuster, S. Riedel, R. Belder, D. Leidner, A. Wedler, and F. Stulp are with the Robotics and Mechatronics Center (RMC), German Aerospace Center (DLR), Wessling 82234, Germany (e-mail: sebastian.brunner@dlr.de; peter.lehner@dlr.de; martin.schuster@dlr.de; sebastian.riedel@dlr.de; rico.belder@dlr.de; daniel.leidner@dlr.de; armin.wedler@dlr.de; Freek.Stulp@dlr.de).

M. Beetz is with the Institute for Artificial Intelligence (IAI), University of Bremen, Bremen D-85748, Germany (e-mail: beetz@cs.uni-bremen.de).

This paper has supplemental downloadable multimedia material available at http://ieeexplore.ieee.org, provided by the authors. The Supplementary Materials contain a video that shows an uncut run of the first part of the "Active Seismic Measurement" mission. This material is 24.4 MB in size.

Digital Object Identifier 10.1109/LRA.2018.2794580

phase. On the one hand, this architecture provides developers means to check the correctness of their task implementations and, on the other hand, the mission control instruments to monitor the mission progress in a detailed manner. Sophisticated analysis of robotic behavior enables the discovery of bottlenecks and critical passages in the overall task sequence. Ultimately, it allows for the optimization of the overall task performance in the context of time and resource usage.

In summary, the contributions of our letter are:

- an architecture for the whole life-cycle analysis of robotic tasks
- a unified framework that implements the proposed analysis concepts
- a case study on a moon analogue mission with a complex robot in an unknown rough terrain
- comparison to other task control frameworks in terms of logging and visualization capabilities, and comparison to profiling frameworks for programming languages

The letter is structured as follows. In the next section, we define the action and resource concept. Section III describes the task analysis concepts of the three different phases in the life-cycle of a robotic mission. In Section IV, we then describe our own implementation of the architecture with focus on *RMC Advanced Flow Control (RAFCON)*. Afterwards, several ROBEX experiments are introduced (Section V), followed by a discussion of the application of all presented methods. The related work is given at the end to allow a detailed comparison of our and related frameworks.

II. Action and Resource Concept

Throughout the letter, an action is regarded as a robotic behavior at an arbitrary level of abstraction, i.e. low level action, capability, skill, subtask or high level task. Every low level action has one code block assigned to it, which is executed on the robot if its action is triggered. Higher-level actions can be composed of lower-level actions, which can also be executed in parallel (for a detailed description see [2]). When the execution of an action is recorded, low-level data accumulates.

As data cannot be interpreted without context, semantics for each action are required. One approach for this is to use ontologies with grounded knowledge. Different types of semantic knowledge are relevant. Most important are the terms *resource* and *action type*. These are needed to define the resources an action needs in order to be executed, and to define which class an action belongs to. In the robotic use case, examples for such action types are navigation, manipulation or object detection actions.

In principle, a resource itself can be anything an action needs to be executed. There are several classes of resources, e.g. robot, domain and task specific resources. Ontology engineering for the task and the domain is a key for the appropriate classification of actions, and thus for the classification of recorded raw data.

III. Task Analysis Architecture

The overall architecture of our whole lifetime analysis is shown in Fig. 2. It is divided into three rows, each of them maps to a certain phase in the life-cycle of the iterative development

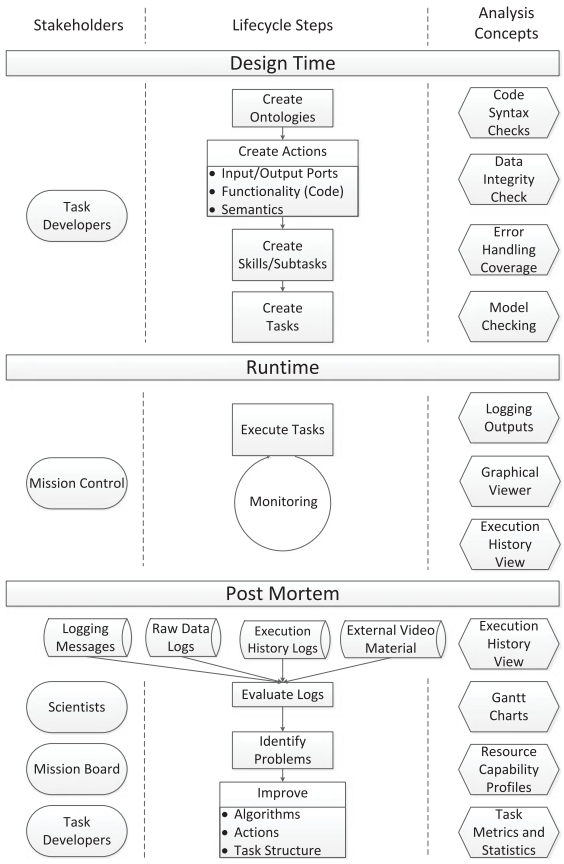

Fig. 2. Whole lifetime analysis architecture overview with all three life-cycle phases including their respective analysis features. This figure derives UML elements from use case, flow, and component diagrams to incorporate the relevant information in one chart.

of a task. In the following, the analysis features are described, which assist the different stakeholders listed in the left column of the figure. In general, these features can be applied both to simulated and real robotic use cases.

A. Design Time

The design time refers to the phase where robot-, task- and domain-specific ontologies as well as actions, subtasks, and the overall mission sequence are created. Several debugging and analysis concepts are appropriate for this phase.

Code Syntax Checks: Static code analysis is used to reveal syntax errors in the code snippet of an action.

Data Integrity Checks: As different actions are parameterized with different data types, consistency for these data types should be guaranteed in every point in time.

Error Handling Coverage: Robotic actions can fail, and they do. Thus, error detection and recovery routines have to be programmed for critical sequences. In general, there are two approaches for error detection: On the one hand, every action itself has to provide exact results and whether it was successful or not. In an error case, the reason for the error also has to be given, e.g., for a grasp action, if the fingers of the gripper did not reach their final position, the grasp contact force did not meet the expected one or if the gripper driver did not answer at all. Apart from this synchronous response, dedicated observers need

to be setup to monitor all critical states of the environment of the robot. Common examples are observers tracking a grasped object and checking that the object remains attached to the gripper, or supervising a navigation action in order to preempt it if a certain time or distance threshold is met.

Apart from being able to detect errors, error recovery routines for each action that is likely to fail have to be chosen (either predefined by the designer of the state machine or planned online). For the asynchronous case, an error handling procedure for each observed state needs to be defined. In a complex system of several hundred actions the creation of dedicated error handling routines for each action does not scale. Thus, reusability of error procedures and modularity, in terms of passing errors to the more abstract levels in a hierarchical action tree and handling them there in a general way, are key features for a scalable architecture. Important exception handling metrics are discussed in Section V.

Model Checking: Certain properties of a system are critical to maintain during a mission: The power for the motors of a quadrocopter for example must not be turned off while the UAV is in the air. Therefore a model checker needs to be integrated, which enables verification that a property holds for the whole runtime. Furthermore, deadlock detection and reachability analysis can be done with this methodology.

B. Runtime

To support the mission control during the execution of a task, we propose two monitoring tools apart from classical logging outputs to the console.

Graphical Viewer: The graphical viewer shows all actions including possible future ones and the currently executed ones. Live data introspection for all data passed to actions as parameters is necessary to keep track of the current mission status.

Execution History View: A graphical visualization of the execution history of all executed actions and their context. It is similar to the stack trace of an *Integrated Developer Environment (IDE)*.

C. Postmortem

The post-mortem analysis represents the largest part of the architecture of Fig. 2. There are many stakeholders interested in the results and the evaluation process takes multiple data sources into account.

Execution History View: The execution trace of all actions must not only be accessible during runtime but also after a mission. The log data of the execution history is the most important data source to create task statistics.

Gantt Charts: With the type and timing information of each action at hand, Gantt charts for arbitrary time windows can be generated.

Resource Capability Profiles: Charts similar to classical *Resource Capability Profiles (RCPs)* can be created due to the fact that the robot knows about the resources each action requires (see Section II). As RCPs are in general not suitable for plotting several resources into the same RCP, we propose a new type of graph. Inverted resource capability profiles can be realized with bar plots *(IRCP-Bars)* instead of scatter plots [3]. RCP inversion means that not the reduction of a resource relative to the total capacity is plotted, but how many units of the resource are in use. The main reason for inverting the graph is that many resources have a maximum capacity of one. Thus, it is more clear to highlight the use of the resource (i.e. the interesting event) and not the availability of it (i.e. the default state). Finally, information retrieval can be improved by scaling and shifting the y-axis differently for each resource. Thus, this graph type is called *Inverted Resource Capability Profile with Shifted-Y-Axis using Bar plots (IRCP-SYA-Bars)*. This graph can be enhanced further by overlaying it with a classical RCP as it is shown in Fig. 7 in Section V.

Task Metrics and Statistics: Different metrics can be calculated with the log data at hand. Example statistics of actions grouped by action type or resource usage are given in Section V-B. Next to statistics about the semantics of states, metrics about exceptions, which occurred in a specific time frame, are also discussed.

IV. Implementation

The architecture presented in the previous section is a general concept and implementation-independent. However, it does place high requirements on the underlying task execution architecture. In this section, we describe our task programming framework RAFCON [2], which was extended so that all these requirements are met, and our analysis framework can be fully exploited.

A. RAFCON for Task Programming

RAFCON is an integrated development environment for programming the autonomous behavior of a robot. The flow control software was developed by DLR-RM. It has already been successfully employed in the space-related *SpacebotCamp* 2015 [1], and in the industrial project RACELab [4]. We showed that the flow control software scales up to systems with more than a hundred processes. Per design, it is middleware-agnostic, which enabled us to integrate the three different middleware *ROS*, *SensorNet* and *Links and Nodes* at the same time [1]. The ability to program error-tolerant and concurrent behavior is deeply integrated into the core design of our tool. For the ROBEX use case we designed state machines with more than 1500 states and 1900 transitions.

B. Execution Engine

Based on hierarchical, concurrent state machines, a behavior programmed with RAFCON holds all technically relevant information to represent the *task flow*. Actions as defined in Section II can be mapped to a state or a hierarchy of states.

The execution engine of RAFCON treats single states, hierarchy states, and whole state machines in exactly the same manner. State machines are also in charge of the *data flow* between individual components [2]. The execution itself can be controlled in a very fine-grained manner. In Fig. 3 possible execution commands for a state machine with three hierarchy levels are shown. Furthermore, the figure shows the type of data that is logged during each execution step.

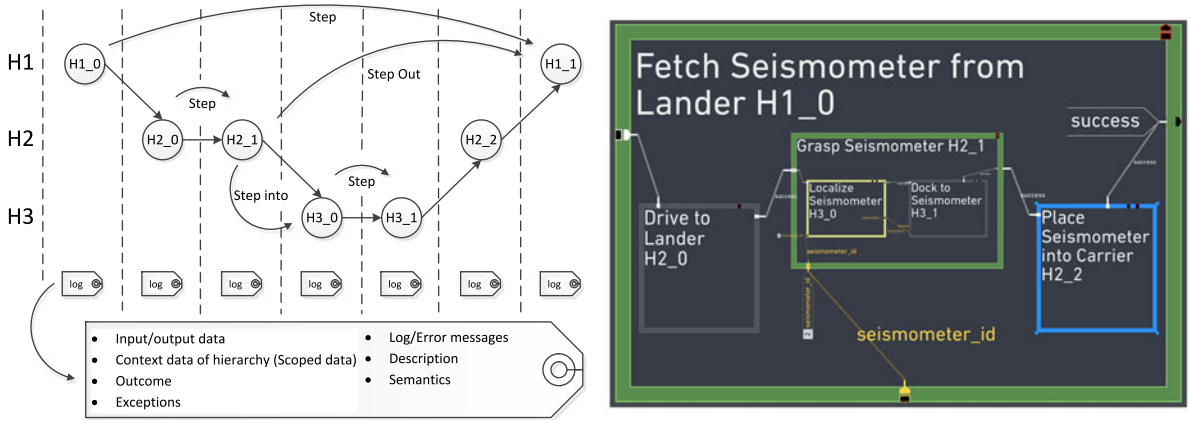

Fig. 3. Left figure visualizes RAFCON's execution and logging architecture using the example state machine on the right side, which is based on the ROBEX use case. It consists of three hierarchy levels (H1—H3). Different execution commands are shown by curved arrows defining a start and an end state. If the execution command represented by the arrow's label is triggered, the execution engine runs every state between the start and the end state. An overview of the data that is logged in every execution step is given.

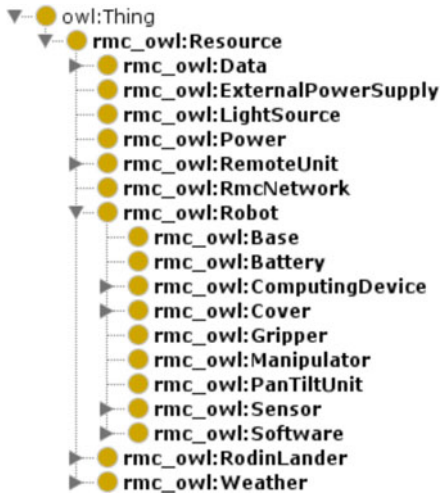

Fig. 4. Part of the resource ontology used during ROBEX.

C. Semantic Logging

During runtime, detailed context information regarding logic transitions and data flows as well as semantic data is logged (see Fig. 3). Moreover, all timing information (i.e. the start times, end times and durations) for each activity is recorded. In the ROBEX use case several hundred megabyte of plain text data was produced every day. To semantically annotate our states, we created two ontologies: one ontology for resources (see Fig. 4) and one for actions types.

The library and hierarchy concept of RAFCON [2] leads to reuse and inheritance of semantic data. This is based on the fact, that *library states* can be reused (i.e. linked) several times into a robotic behavior but only have to be annotated once. Thus, a developer only has to annotate a fraction of the whole set of states (in the state machine of Section V, less than 5% of all states had to be annotated). In general, both the amount of semantic information per action and the coverage of the state machine annotation is adaptive. Thus, the developer is able to make a conscious decision concerning the tradeoff between manual annotation overhead and the degree of the semantic structure and classification of raw data.

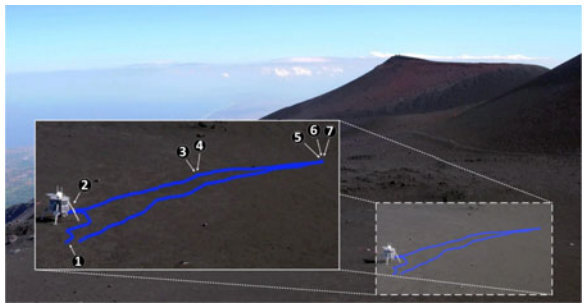

Fig. 5. Semi-automated Google Earth export of the trajectory of the LRU on a mission during the ROBEX project on Mt. Etna, Sicily. Some poses are labeled with numbers.

D. Implementation Details of Task Analysis Features

RAFCON implements each of the task analysis features given in Section III. To calculate the error handling coverage of a state machine, all states whose "aborted" outcomes (triggered in the case of errors, see [2]) are connected to another state are regarded as covered.

For model checking, we integrated *DIVINE* [5] into our framework [6]. Every state has to be annotated with *DVE* syntax, so that the definition of the behavior of the state is compatible with DIVINE. The RAFCON state machine structure together with these annotations are used to build a transition system. Finally, DIVINE can validate arbitrary system properties, which can be defined via *Linear Temporal Logic (LTL)* [5], for this transition system. DIVINE uses partial order reduction for state space reduction and can distribute model checking tasks to multiple nodes.

A foldable *Execution History Tree View* shows all executed states in their respective hierarchies together with all the context data during state execution. The execution history is also accessible after runtime, but in a lightweight GUI optimized for large data sets.

We employ the plotting library *Plotly* [3] for displaying Gantt charts and RCPs in an interactive way, which allows for panning and zooming inside the diagram. Arbitrary information can be rendered at the mouse position while the mouse is hovering over

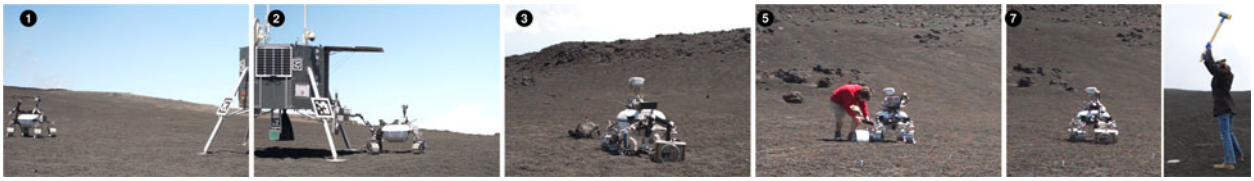

Fig. 6. Scenes from the moon-analogue experiment on Mt. Etna, Sicily in Italy. The labels refer to those of Fig. 5.

a certain part of the chart. This is used to show the context data of the selected state. Multiple task executions over several sessions can be grouped and analyzed together. Also the visualization of the runs of several robots can be performed.

For implementation details about the remaining analysis features "Code Syntax Checks," "Data Integrity Checks," and the "Graphical Viewer" we refer to [2].

V. Case Study

We evaluated our task execution profiling in a fully autonomous *moon analogue* mission, to show the applicability of the approach in a real robotic scenario.

A. Experiment Description

The goal of the planetary exploration experiment in ROBEX is to analyze the geologic composition of a remote planet with seismic instruments. The task of the rover is to fetch the seismic instrument from a landing unit and place it on the ground at a certain position. Furthermore, the robot must level the ground and produce a test impulse to ensure the correct deployment of the instrument. Once the instrument is correctly deployed, it measures a remote impulse and the robot subsequently places it at the next measurement position. The overall mission consisted of several submissions: *Active Seismic Measurement (ASM)*, *Seismic Network* and *Sample Return*. In the first mission, the rover had to drive to predefined spots and had to perform a seismic measurement with one instrument. During the second mission, the rover had to deploy a network of four different measurement units. For the last mission the rover had to take a soil sample with a shovel and return it to the base.

Fig. 5 shows an overview image of the ASM mission at the test location on Mt. Etna. We selected some key frames (1–7) from the mission to illustrate the execution logging. Fig. 6 shows real scenes from the mission for the very same key frames. The robot starts in front of the landing unit (1) and drives to the seismic instrument hand-over position. There the rover docks to the seismic instrument (2) and waits for the landing unit to release the instrument. Once the rover has stored the instrument on its back, it drives to the first deploy location and places the seismometer on the ground. To ensure the correct function, the robot levels the ground with the instrument (3). Once the instrument is correctly deployed, the rover waits for an external seismic impulse, which the instrument records (4). The rover repeats the process (6 and 7) at a second location and drives back to the landing unit. During the mission, an operator had to change the battery (5).

TABLE I
Statistics of the State Machines for the Main Robex Missions

State machine:	State Count	Tran-sition Count	Max Depth	Percentage of States with EH	Max EH Depth	Average EH Depth
ASM	1532	1996	8	36.62	2	1.02
Seismic Network	1455	1905	8	34.64	2	1.03
Sample Return	947	1184	7	28.51	2	1.05

Columns Two and Three Show the Total Number of States and transitions. The next column gives the maximum number of hierarchies a state can reside in. The fifth column gives the percentage of all states covered with *Error Handling (EH)* routines. Sometimes also backup procedures for error recovery behavior are needed, which creates a chain of error handling routines. The length of such a chain is called the error handling depth.

B. Experiment Analysis

In the following, we showcase how the proposed analysis concepts were used to support the mission on Mt. Etna.

1) Design Time Analysis: To create the autonomous behavior of the LRU we made extensive use of the code syntax and data integrity checks. Thus, many errors could be eliminated before runtime. To make a statement about the robustness of the task implementation, Table I can be taken into account. It shows that, although the state machines are big, the error handling coverage is still at more than 28 percent of all states for all missions. This and the fact that some states even had nested error recovery procedures made the state machine robust.

2) Runtime Analysis: Next to the benefit of logging outputs with filterable severity levels, we made extensive use of the online execution history view. If any anomaly occurred, we could pause the execution engine and examine the current context data of the state machine. Especially having access to every executed movement of the manipulator, with its stiffness and damping factors, and interpolator parameterization, helped us to find errors quickly.

Furthermore, the graphical viewer of the state machine supports the mission control to keep track of the current internal state of the autonomous behavior. The right side of Fig. 3 shows the execution of an example state machine used during ROBEX. Green highlights the currently active states, light yellow the last executed one, and blue the currently selected state. The logic flow is shown in gray lines and the data flow in dark yellow lines. Next to the data ports the current data is visualized. For example, the mission control immediately sees that the current value of the "seismometer_id" has the value 2.

3) Postmortem Analysis: From the logs gathered during the experiment, we created two different types of visualizations.

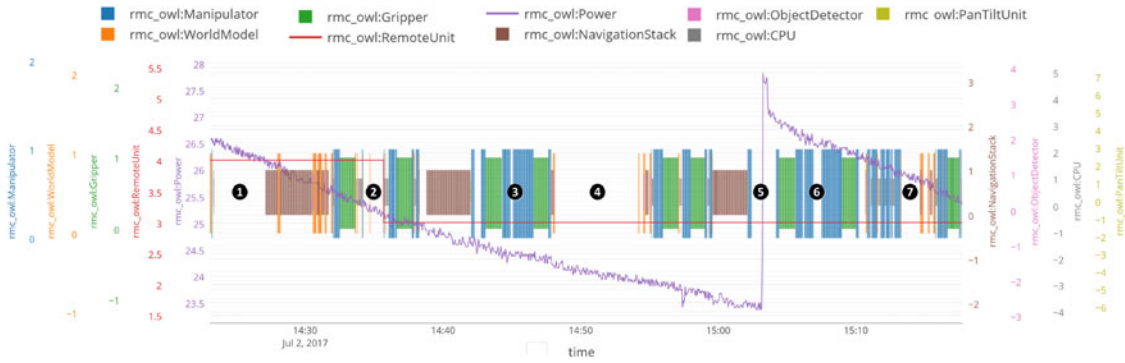

Fig. 7. IRCP-SYA-Bars diagram of the the state machine created for the *Active Seismic Measurement (ASM)* mission of ROBEX. The labels show milestones and interesting events of the mission and are the same labels as those of Fig. 5.

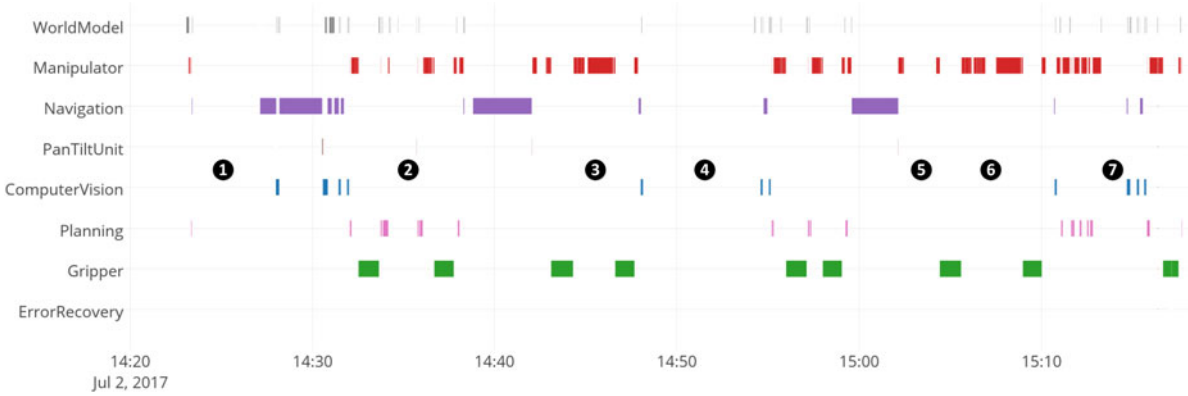

Fig. 8. Gantt diagram of the ASM mission. All executed actions are grouped by their action type (shown on the y-axis). The labels refer to those of Fig. 5.

TABLE II
METRICS FOR ACTIONS GROUPED BY ACTION TYPES FOR THE ASM STATE MACHINE OF THE ROBEX MISSION

Action Type	Total Call Count	Spent Time	% of Total Time	Max Time	Min Time	Time per Call
World Model	106	42.53	1.50	2.69	0.12	0.40
Manipulator	100	624.73	21.98	82.34	0.10	6.25
Navigation	31	1164.57	40.98	192.52	0.10	37.57
Computer Vision	30	94.24	3.32	15.84	0.15	3.14
Planning	19	101.82	3.58	17.14	1.34	5.36
Gripper	12	506.54	17.82	69.93	0.60	42.21
PanTilt Unit	6	4.09	0.14	2.79	0.09	0.68
Error Recovery	1	0.02	0.00	0.02	0.02	0.02

TABLE III
THE TEN EXCEPTIONS THAT OCCURRED MOST OFTEN DURING THE ROBEX MISSIONS DURING THREE WEEKS (BETWEEN 2017-06-14 AND 2017-07-06)

State Name	Excep-tion Count	% of Total Count	Caught	Different Origins
get first object from list	91	16.73	59	17
plan joint move	62	11.40	6	31
open	41	7.54	29	15
close	39	7.17	36	14
check transform validity	35	6.43	26	10
get submap id	25	4.60	23	3
plan task	18	3.31	7	5
detect object	18	3.31	5	8
move rel cartesian impedance	15	2.76	12	9
load world	13	2.39	0	7

The columns from left to right show: States in which the exception occurred; total exception count; relative exception count; number of caught exceptions; at how many different locations this state was included.

Fig. 7 shows a diagram of the most important resources the rover used during the mission: The Jaco 2 [1] manipulator, the docking interface (i.e. the gripper) on the manipulator, the navigation stack, the world representation, the seismic instruments and the power voltage of the rover. The diagram shows that during the initialization (1), the docking interaction with the landing unit (2), the battery exchange (5), and the seismic measurement (4 and 7), most of the resources of the rover are idle. Additionally during the battery exchange (5), the power voltage is replenished to 27.8 volts. While the rover places and levels the instrument (3 and 6), the diagram shows heavy usage of the manipulator and the docking interface. Fig. 8 presents a Gantt chart of the execution times of the individual software and hardware components. The plot shows that most of the execution time was spent during manipulation while moving either the manipulator (red) or docking or undocking the seismic instrument with the docking interface (green).

TABLE IV
FEATURES SUPPORTED BY DIFFERENT TASK PROGRAMMING SOFTWARE

Task Software	Data Integrity	Error Handling Coverage	Graphical Live Data	Execution History Online	Logging Outputs	Semantic Logging	Execution History Offline	Gantt Charts	RCPs	Task Statistics
Smach			x		x					
ROS Commander			x		x					
Xabsl	x		x		x					
Flexbe	x		x	x	x		x			x
CRAM+openEASE					x	x	x	x		x
RAFCON	x	x	x	x	x	x	x	x	x	x

As already highlighted, the created data and metrics can be used to identify bottlenecks of the overall task. In Table II, it can be seen that manipulator, navigation and gripper actions took most of the time. Furthermore, for navigation and gripper actions, the average time per call is the highest. An explanation for the navigation action runtimes is simply the covered distance that the rover traveled during the mission. Obviously, the gripper itself is the biggest bottleneck of the mission, as only 12 calls take more than 17.82% of the overall runtime. One reason for this is that a tight form closure between the gripper and the remote unit is needed for the manipulation tasks. At the same time, the gripper has to be very lightweight in order not to reduce the overall payload of the manipulator. Thus a high mechanical transmission was inserted which reduces the speed of grasping.

Furthermore, in Table III, we see that many errors occurred during our mission days on Mt. Etna. Many of them were caught, which means that the error handling procedures (see again Table I) in the task implementation targeted the correct spots. Both the total number of exceptions and the number of uncaught exceptions yield interesting results. On the one hand, the object detection was not robust enough. Both the "get first object from list" and the "detect object" actions belonged to the class of object detection routines. Moving the robot slightly and re-detecting the object, or removing bugs or butterflies (on Mt. Etna there are many of these in June) from the object surface helped to continue the mission. On the other hand, the planning of a motion was not properly encapsulated in error recovery strategies, as can be seen by the low caught rate of the states "plan joint move" and "plan task". As we use a statistical planner based on sampling, simple replanning often resolved the issue. The "load world" action was never caught, but this is unproblematic as the action only returned with an error if there were syntax errors in the world description or the world representation node was not started yet. These errors could be avoided by respecting spelling and syntax rules or following the correct robot system startup procedure.

VI. RELATED WORK

In the following, we give a comparison of RAFCON to robotic task programming frameworks on the one hand and a comparison between our tool and profilers of high-level programming languages on the other hand.

A. Comparison With Related Task Programming Frameworks

In principle, logging and analysis of task-related data for robotic applications is an important topic. There are many software tools available that try to tackle this topic as well. *CRAM* [7] in combination with *openEASE* [8], e.g. offers powerful semantic task logging features that enable to replay the high level actions executed by a robot. The semantic knowledge is, as in our approach, defined in ontologies. *openEASE* is not only used for visualization but also as a mean to retrieve knowledge matching an arbitrary pattern via Prolog. Unfortunately, this framework does neither feature design time and runtime analysis possibilities nor debugging support related to data handling and data integrity.

ROS Commander [9], *XABSL* [10], *Smach* [11], and *Flexbe* [12] are also examples for behavior programming frameworks that offer usable concepts. Unfortunately, the development of ROS Commander and XABSL discontinued several years ago. Furthermore, they lack important postmortem analysis features. In Smach, although widely used, the task logging is restricted to recording console output only. Flexbe on the other hand offers quite powerful logging and analysis features but does not include the recording of semantic knowledge based on ontologies.

Based on the analysis features of Section III, we performed a comparison between RAFCON and all mentioned task programming tools. The results are shown in Table IV. It shows that, except CRAM, no other framework supports as fine-grained task logging, including semantic data, as RAFCON. Most of the times, none or only some of the features mentioned above are supported. Especially concerning fine grained logging capabilities and error statistics, our task programming tool outperforms all other frameworks.

As we created our architecture and enhanced RAFCON for a space-related mission (see Section V), we would like to highlight NASA's effort in this field as well. From simple event logging frameworks with online modifiable severity levels [13] to timeline and resource capability profile plots [14], logging and analysis of mission critical spacecraft data always played a central role for their software architectures. Recently NASA also enhanced their *Europa* [15] framework for integrated planning and scheduling with features to visualize *Gantt* charts, action details, action violations and solver statistics, i.e. statistics of (generated) action sequences. The framework focuses on the creation of plans and does not provide much support for

TABLE V
FEATURES SUPPORTED BY DIFFERENT PROFILING FRAMEWORKS

Profiler	Total Call Count	Total Spent Time	Time per Call	Time per Total Spent Time	Max / Min Time	Total Exception Count	Relative Exception Proportion	List View of Called Entities	Call Graph	Suitable for Task Profiling
Valgrind + KCachegrind	x	x	x	x		x	x	x	x	
Python "profiling"	x	x	x	x				x		
cProfile + graphViz	x	x	x	x				x	x	
JProfiler	x	x	x	x	x	x	x	x	x	
RAFCON	x	x	x	x	x	x	x	x	x	x

runtime and postmortem analysis. Finally, NASA recently created a telemetry visualization GUI called Open MCT [16]. However, it is aimed at the visualization of logged telemetry data, not the logging itself.

B. Comparison with Profilers of Programming Languages

When designing and implementing the architecture for whole life-cycle analysis, we tried to keep close to the concept of profilers for high-level programming languages. Well known examples in this area are *Valgrind* [17], *JProfiler* [18], *cProfile* [19] together with *Graphviz* [20], and the *profiling* package of Python [21].

In Table V, we give a feature list of what common profilers normally offer and compare different profiling frameworks using these features. The table shows that RAFCON exhibits all necessary profiling capabilities. However, to the best of our knowledge, nobody did profiling on pure task level before.

VII. CONCLUSION

In this letter, we proposed an architecture for the whole life-cycle analysis of autonomous, robotic tasks and demonstrated it by a case study in the context of the ROBEX project on Mt. Etna, Sicily. We showcased how to identify critical bottlenecks in our mission and how to gather precious information about the robustness of our task. Furthermore, we compared existing task programming frameworks and profiling tools with RAFCON, showing that our framework has the most complete feature set and scales very well to prolonged autonomous tasks. Possible future work is the integration of a prediction step between online and postmortem analysis in order to run online anomaly detection algorithms. Also a simulation step for early unit and integration testing before runtime would be a powerful enhancement to our architecture. Moreover, this work will be, like our flow control software itself, released as open source in the near future. Finally, we will use the whole framework in an industry 4.0 use case as well.

REFERENCES

[1] M. J. Schuster *et al.*, "Towards autonomous planetary exploration: The lightweight rover unit (LRU), its success in the spacebotcamp challenge, and beyond," *J. IntellRobot. Syst.*, Nov. 2017. [Online]. Available: https://doi.org/10.1007/s10846-017-0680-9

[2] S. G. Brunner, F. Steinmetz, R. Belder, and A. Doemel, "RAFCON: A graphical tool for engineering complex, robotic tasks," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Deajeon, South Korea, 2016.

[3] "Plotly," 2017. [Online]. Available: https://plot.ly/. Accessed on: Aug. 30, 2017.

[4] F. Steinmetz and R. Weitschat, "Skill parametrization approaches and skill architecture for human-robot interaction," in *Proc. 2016 IEEE Int. Conf. Autom. Sci. Eng.*, Fort Worth, TX, USA, Aug. 2016, pp. 280–285.

[5] J. Barnat *et al.*, "DiVinE 3.0—An explicit-state model checker for multithreaded C & C++ Programs," in *Computer Aided Verification (CAV)*, vol. 8044. New York, NY, USA: Springer-Verlag, 2013, pp. 863–868.

[6] M. Vilzmann, "Mission planning and verification for autonomous unmanned aerial vehicles," M.S. thesis, Clausthal Univ. Technol., Clausthal-Zellerfeld, Germany, 2016. [Online]. Available: http://elib.dlr.de/115188/

[7] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM—A cognitive robot abstract machine for everyday manipulation in human environments," in *Proc. IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, Taipei, Taiwan, Oct. 18–22 2010, pp. 1012–1017.

[8] M. Tenorth, J. Winkler, D. Beßler, and M. Beetz, "Open-EASE—A cloud-based knowledge service for autonomous learning," in *KI – Knstliche Intelligenz*. Berlin, Germany: Springer-Verlag, 2015.

[9] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. C. Kemp, "ROS commander (ROSCo): Behavior creation for home robots," in *Proc. IEEE Int. Conf. Robot. Autom.*, Karlsruhe, Germany, May 2013, pp. 467–474.

[10] M. Loetzsch, M. Risler, and M. Juengel, "XABSL—A pragmatic approach to behavior engineering," in *Proc. IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, Beijing, China, 2006, pp. 5124–5129.

[11] J. Bohren and S. Cousins, "The SMACH high-level executive," *IEEE Robot. Autom. Mag.*, vol. 17, no. 4, pp. 18–20, Dec. 2010.

[12] P. Schillinger, S. Kohlbrecher, and O. von Stryk, "Human-Robot collaborative high-level control with an application to rescue robotics," in *IEEE Int. Conf. Robot. Autom.*, Stockholm, Sweden, May 2016.

[13] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, "Software architecture themes in JPL's mission data system," in *Proc. IEEE Aerosp. Conf.*, Big Sky, MT, USA, 2000, vol. 7, pp. 259–268.

[14] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, "Integrated planning and execution for autonomous spacecraft," in *Proc. IEEE Aerosp. Conf.*, Snowmass at Aspen, CO, USA, 1999, vol. 1, pp. 263–271.

[15] J. Barreiro *et al.*, "EUROPA: A platform for AI planning, scheduling, constraint programming, and optimization," in *Proc. 22nd Int. Conf. Autom. Plan. Scheduling*, Atibaia, 2012.

[16] NASA, "Open MCT," [Online]. Available: https://nasa.github.io/openmct/. Accessed on: Aug. 30, 2017.

[17] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Program. Language Des. Implementation*. New York, NY, USA: ACM, 2007, pp. 89–100.

[18] G. Gousios and D. Spinellis, "Java performance evaluation using external instrumentation," in *Proc. 2008 Panhellenic Conf. Inform.*, Samos Island, Greece, Aug. 2008, pp. 173–177.

[19] "cProfile," 2017. [Online]. Available: https://docs.python.org/2/library/profile.html. Accessed on: Aug. 30, 2017.

[20] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz—Open source graph drawing tools," in *Proc. Graph Drawing: 9th Int. Symp.*, P. Mutzel, M. Jünger, and S. Leipert, Eds. Berlin, Heidelberg: Springer, 2002, pp. 483–484.

[21] What! Studio. 'Python Profiling. 2017. [Online]. Available: https://github.com/what-studio/profiling. Accessed on: Aug. 30, 2017.