Continuous Regression Testing for Component Resource Utilization

Johannes Wienke[1] and Sebastian Wrede[1]

Abstract— **Unintended changes in the utilization of resources like CPU and memory can lead to severe problems for the operation of robotics and intelligent systems. Still, systematic testing for such performance regressions has largely been ignored in this domain. We present a method to specify and execute performance tests for individual components of component-based robotics systems based on their component interfaces. The method includes an automatic analysis of each component revision against previous ones that reports potential changes to the resource usage characteristics. This informs developers about the impact of their changes. We describe the design of the framework and present evaluation results for the automatic detection of performance changes based on tests for a variety of robotics components.**

I. INTRODUCTION

Nowadays, most robotics and intelligent systems are complex software systems. In order to fulfill increasingly complicated missions, these systems often comprise a multitude of hardware and software components and are developed by larger and often distributed teams. With the emergence of common frameworks (e.g. ROS [1] or YARP [2]), systems often contain off-the-shelf components, with implementations outside of the control of system integrators. This makes the task of maintaining such a system and ensuring its proper functionality a challenge. Nevertheless, many applications require a high reliability and availability of the deployed systems in order to meet the users' demands and to ensure their safety. Consequently, verification techniques become more important during development to ensure the proper functionality of individual components and the integrated system. Apart from common software engineering methods like unit testing at class-level and integration testing at component-level, simulation is a common method applied in robotics to verify the functionality robotics applications at system level [3].

Most of the currently applied testing techniques aim at the delivery of the promised functionality of the system and its components. However, an aspect that has been largely ignored so far in this domain is a systematic verification of the required computational resources. Unplanned changes in the consumption of resources like CPU time, memory, or network bandwidth can lead to various undesired effects on the system, e.g., increased power consumption, delays, reduced accuracies, or component crashes. Depending on the affected subsystems and the application domain, the outcomes of these effects can range from the unnecessary consumption of energy or vanishing user acceptance to severe injuries in case of safety-critical systems. Therefore, the computational performance of robotics systems needs to be closely observed and suitable tools are required to detect performance regressions as soon as possible during development. Such tools need to be applicable for non-experts users, should perform their task automatically for each new revision of a software component, and they should integrate with modern development workflow (e.g. continuous integration (CI) servers [4]) to foster wide adoption and to ensure a high coverage of systems and component revisions.

In this work we present a method for generating and analyzing performance tests for individual components of robotics and intelligent systems. It aims to automatically detect performance regressions introduced by code changes as soon as possible. The method exploits the fact that most current systems are composed of components which communicate via a middleware. A new performance testing framework tests the components using their component interfaces and provides methods to automatically detect and report changes in the performance characteristics. Tests are based on an event generation language which defines abstract test cases that are instantiated for different parameter combinations to explore the runtime behavior of the components under different loads. The framework is designed to integrate into automated build processes. We report on the design of the framework and evaluation results on our systems.

II. RELATED WORK

In contrast to robotics, systematically testing software for performance regressions is a common practice in other disciplines, most notable being large scale enterprise systems and website operation. These disciplines are origin of the "Application Performance Management" (APM) method [5], [6], which combines different practices and tools with the aim to detect performance issues before becoming a problem in the field. In this area, application performance is usually defined along two dimensions of key performance indicators (KPIs): service-oriented (e.g. response times, number of requests) and efficiency-oriented (e.g. CPU) KPIs [6], where efficiency-oriented KPIs match our general motivation. Testing performance in these systems is usually performed on a much coarser-grained level with the whole system being deployed for testing as a monolithic unit. Tests are often performed based on mimicking or abstracting the human users of the systems (e.g. through http interactions) and can last up to several hours or days [5].

A recent survey by Jiang and Hassan [7] provides a good overview on how research addresses the issue of performance

[1] Research Institute for Cognition and Robotics (CoR-Lab) and Cluster of Excellence Cognitive Interaction Technology (CITEC), Bielefeld University {jwienke,swrede}@techfak.uni-bielefeld.de

testing of large-scale systems. The authors separate the testing process into three successive steps: test design, execution and analysis and categorize publications along several axes inside each step. The review does not mention any work that specifically focuses on individual components as the unit of testing. Instead, most approaches follow the APM idea of focusing on the complete system.

Following the proposed separation, several distinct methods to design performance tests exist. Tools like *Apache JMeter* [8] and *Tsung* [9] focus on testing via network protocols like HTTP or XMPP and provide methods to generate tests for these protocols. Often, recording capabilities exist to generate interactions based on prototypes and loops and parallel execution can be used to generate extended loads using an abstract specification of the interactions. In JMeter, most interactions are primarily GUI-based, while Tsung uses an XML configuration file and command line utilities for defining tests. Other tools like *Locust* [10], *NLoad* [11], *The Grinder* [12], and Chen *et al.* [13] use the programming language level to define load tests while tools like *Gatling* [14] are in between these categories by generating code from exemplary executions. In contrast, Da Silveira [15] presents an approach which uses a Domain Specific Language (DSL) to formulate performance tests inside the model-based testing paradigm. Generally, the presented approaches usually provide a way to structure the performance or load test into distinct units like test cases or test phases.

For executing tests, frameworks have the duty to generate the load and to log metrics during the test. Depending on the framework, load can be generated from one or several hosts, e.g. [8]–[10]. Most IP-based frameworks automatically log metrics like response times for the issued requests. Additionally, some of them incorporate ways to also log resource usage on the tested systems, e.g. [8], [9], [12].

For analyzing the results of performance tests with the aim to automatically detect performance regressions, several methods have been proposed. One common method is the use of control charts [16]–[18]. However, control charts assume normal distributions for the measured values, which is usually not the case for performance counters like CPU usage under varying load. Another category of approaches exploits the fact that several performance counters in a test run are usually correlated and changes in these correlations could indicate a performance regression. Moreover, correlation might be used to reduce the amount of counters that needs to be analyzed. Foo *et al.* [19] and Žaleźničenka [20] implement this approach by applying association rule learning techniques while Shang *et al.* [21] present a method based on clustering and regression. Additionally, Malik *et al.* [16] present another clustering and a PCA-based approach.

Generally, the existing work mostly focuses on performance testing integrated systems. While such tests are also desirable for robotics and intelligent systems, they are much harder to set up and maintain due to the complex interactions of robots with the real world and the non-standard interfaces in contrast to e.g. HTTP. Moreover, performance regressions detected in such integrated tests are harder to point down to individual components for fixing the issues. Therefore, testing performance for individual components provides a parallel, and currently better applicable method in robotics and intelligent systems.

Finally, in addition to the presented data-driven methods for detecting performance regressions, there is also research on prediction resource consumption based on software models. For instance, Becker *et al.* [22] present the quite popular Palladio Component Model, which allows to model complete software architectures with respect to performance-relevant aspects. Comparable approaches have been reviewed in Koziolek [23]. In case a complete model of the system exists, performance regressions can be derived from the model. However, these models often do not exist or are out of sync with the real system. Additionally, specifying the resource usage of a manually implemented or third-party component is a non-trivial task and therefore, models often present a level of abstraction that might miss certain performance degradations that are practically important or noticeable.

III. CONCEPT

Most current robotics systems are constructed as distributed component-based systems where individual components implement isolated aspects of the system functionality [24]. A component is thereby a specification of an interface of how potentially multiple different implementations interact with the rest of the software system. While the interface usually remains relatively stable, the implementation might change frequently and therefore also its performance characteristics. For our case, we assume that a component's interface is defined and realized in terms of communication patterns [25] and data types of a middleware framework like ROS or YARP. With the ongoing adoption of a limited set of such frameworks, a reasonable part of components is reused across different systems. Additionally, current systems are often maintained by multiple persons and no single person has in-depth knowledge of all components which form the system. Therefore, we think that it is viable to implement performance testing on a per-component basis instead of or in addition to the complete system because:

1) testing a complete robotics system for performance regressions in an automated fashion is very hard to achieve due to the interaction with the real world like speech-based dialog or computer vision problems.
2) the middleware-based component interface allows to write performance tests that are quite stable during component and system evolution.
3) detected performance regressions can be attributed easily to the component as a code unit.
4) component developers have the best knowledge about their components and the expected loads and behaviors. Therefore, developers can test the complete range of functionality and loads and not only the requirements of a single target system.
5) components often exist longer than individual systems and therefore test results should be available and comparable even if systems change.

Therefore we present a framework to specify, execute and analyze performance tests for individual components of robotics systems. These tests are maintained alongside the component in a similar fashion to unit tests[1]. This ensures that tests are kept up to date with the component by the component developers and test results are immediately available after component changes.

Depending on the connectivity of a component with the remaining system or the underlying operating system and hardware, testing via the middleware interface can be more or less complicated. E.g. a controlling state machine usually communicates with many other components in the system. Therefore, it is hard to test it in isolation. While it is possible to test such a component (e.g. by implementing mock components for the tests), our approach primarily targets what Brugali and Scandurra [24] call "vertical components", which capture isolated domain knowledge in a functional area and contribute most to reuse.

IV. REALIZATION

In the following subsections we describe the realization of our approach. According to Malik *et al.* [16], a common load or performance test (terms are often used interchangeably [7]) consists of "a) test environment setup, b) load generation, c) load test execution, and d) load test analysis". We agree with this and the following descriptions of our framework follow this separation (with a changed order). The framework has been developed using the RSB middleware [26], but the concepts can easily be applied for other common frameworks like ROS or YARP, as long as tools exist to record and replay communication.

A. Load Generation

For vertical components, we assume that the resource demands of the component at runtime are to a large extent related to the communication a component is exposed to. For instance, a face detection algorithm might impose higher or lower CPU usage depending on the rate and size of images it received via the middleware. Similarly, a person tracker's CPU and memory usage might relate to the number of person percepts it receives via its communication channels. Therefore, generating load in terms of middleware communication provides a way to abstract the ongoing development changes inside the component while enabling to test the aspects of components that are relevant to its use inside a robotics system.

In order to generate such middleware-defined loads we have implemented a testing framework which allows to specify middleware interactions with components via a Java API. We have chosen Java as our target language as it provides the required performance to generate heavy loads (e.g. in contrast to Python) while consisting of a relatively easy to use programming language and environment (e.g. compared to C++) which is usable for most developers.

[1]Either inside the component's source code repository or in a closely coupled one.

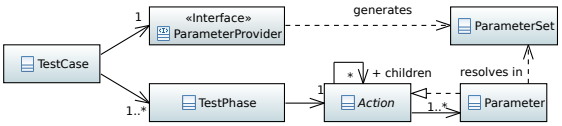

Fig. 1. Structure of the testing API.

Inside this framework we define a performance test to consist of multiple *test cases* (cf. Figure 1. Each test case consists of one or more *test phases* and a *parameter provider*. A test phase is a named entity which consists of a tree of parameterized *actions* to perform via the middleware which specify the actual interaction of the test with the component. These actions have variable *parameters*, like e.g. a sending rate for messages or a number of faces to include in a face detection message. The parameter provider generates *parameter sets* which specify the actual values for all variable parameters. When executing the test case, the action tree is executed sequentially for all parameter sets, thereby generating different load levels on the component.

Parameters allow to specify the load profile of tests, while actions specify the semantics of the interaction. Such parameters can for instance be:

- communication rates
- number of generated messages
- different data sizes
- sets of pre-computed control/data messages
- middleware communication channels

By extracting these parameters from the actual definition of the interaction we gain several benefits:

- The influence of these parameters on the component can be systematically analyzed.
- By changing parameter sets, different test granularities can be achieved.
- Test cases can be reused across different, functionally comparable components by changing parameter sets.

Test phases provide the ability to group operations to perform with the component under test, which are identifiable for a later analysis step. As test phases are executed sequentially inside each test case for all parameter combinations, this allows to define transactions in case a component like a database requires a defined protocol.

The actions that can be performed in order to interact with the component under test form a limited specification language suitable for the needs of performance testing. Each action generally is a function that takes the current parameter set as the input and optionally returns a result, which may be processed by parent actions. We have identified and implemented the actions shown in Table I as a result of testing our own components. Performance tests are created by forming a tree of these actions and potentially user-defined ones for specific use cases. We have provided specific support for generating variable data for our middleware based on the parameters as this is a very common task. The `ProtobufData` action allows to construct Protocol Buffers [27] data (which is primarily used in RSB) from

TABLE I
IDENTIFIED ACTIONS FOR PERFORMANCE TESTING

Data	
Parameter	Resolve a value from a parameter set
StaticData	Resolve to a pre-defined, static value
Flow	
Sequence	Execute actions sequentially
Loop	Loop an action n times or indefinitely
Parallel	Execute multiple actions in parallel
WithBackground	Execute one main action with multiple background actions. Background is interrupted when the main action finishes.
Timing	
Sleep	Sleep for a specified time
LimitedTime	Execute an action up to a time limit
FixedRate	Execute an action at a fixed rate
Middleware (RSB)	
InformerAction	Send an RSB event (message)
RpcAction	Call an RPC server method
WaitEvent	Wait for an event to arrive
BagAction	Replay recorded communication
DynamicEvent	Construct an event (for Informer or RPC action)
ProtobufData	Generate protocol buffers payloads from parameters

template messages by scaling (repeated and string) fields based on parameters. For this purpose, API users provide data generators for the individual items. Additionally, the `BagAction` provides a method to replay pre-recorded data, optionally with modulations like speed or channel selection. If a user requires further actions or methods for generating test data, the action tree can be extended with custom implementations. Figure 2 visualizes the action trees that have been used to construct a test for a leg detector component.

Each test case is equipped with a parameter provider which generates one or more sets of parameter combinations. Each parameter itself is a programming language object and has a printable name for the analysis. We provide two implementations of parameter providers: a table, where the user manually specifies the row values, and a Cartesian product, where combinations of individual parameter values are created, optionally with constraints. For easily specifying the constraints, scripting languages like Groovy can be used. Since parameters will be reported during test execution using the middleware, they must be serializable by the middleware.

B. Environment Setup

For executing performance tests, the API contains a test runner. The first step of this test runner is to set up the test environment based on a configuration file, which specifies the following aspects:

- locations of utility programs required by the testing API
- middleware configuration
- processes which act as a test fixture (e.g. daemons, mock components)
- component processes to test
- test cases to execute and their parameter providers (references to Java classes)

```
Legdetector Test
 ├─TestCase smoke
 │  └─TestPhase fire
 │     └─FixedRateTiming
 │        ├─StaticData 10000 // length (ms)
 │        ├─Parameter scanFrequency
 │        └─InformerAction
 │           └─DynamicEvent
 │              ├─StaticData rst.vision.LaserScan
 │              ├─StaticData /sensors/laserscan/
 │              └─GenerateData
 │                 └─Parameter numberOfPersons
 │           └─StaticData rsb.Informer
 ├─TestCase real-recording
 │  └─TestPhase replay
 │     └─RecordedTimeBagReplay
 │        └─Parameter tideFile
 └─TestCase recording-modulated
    └─TestPhase replay
       └─FixedRateBagReplay
          ├─Parameter tideFile
          └─Parameter scanFrequency
```

Fig. 2. Structure of a performance test for a component which detects legs in a laser scanner. The test consists of three distinct test phases.

Using this configuration, the initialization of the test environment is performed in the following steps:

1) Configuration of the middleware to ensure that test execution is isolated from the remaining system.
2) Creation of a temporary workspace for the test execution. The workspace is used as the working directory for executed processes and stores intermediate logs which can be retained for debugging.
3) Start of all defined fixture processes. These could be daemons required for the middleware, database services used by the tested component etc.

C. Test Execution

See Figure 3 for a visualization of the following aspects.

1) Orchestration: After the environment setup, the configuration is used to instantiate and execute the performance test. First, the configured test case and parameter provider instances are created and a static validation for invalid parameter references is performed. If validation succeeds, the defined components to test are started. While usually only a single component is started, it is also possible to test a combination of components in cases where such a constellation is easier to test than an isolated component due to the required interactions. After starting all components, the test cases are executed sequentially. Inside each test case, the defined test phases are executed for all parameter sets returned by the parameter provider. Finally, all started components and the test fixture are terminated. We have decided to use a single execution of the component processes without intermediate restarts, e.g. for each parameter set. On the one hand, test runs require more time with component restarts and on the other hand, most robotics components usually operate for a longer time without restarting and artificial restarts would make it harder to detect performance issues like memory leaks, which slowly build up over time.

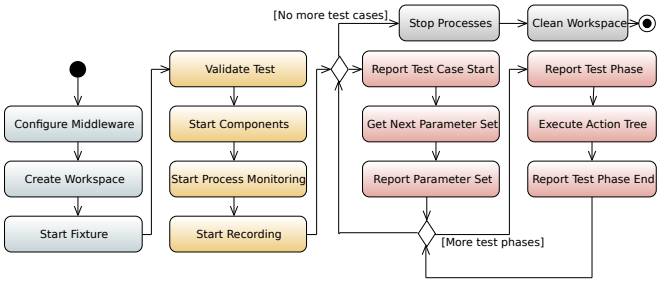

Fig. 3. Visualization of steps performed to execute a performance test. Cyan: environment setup, orange: test setup, red: test case execution, gray: clean up.

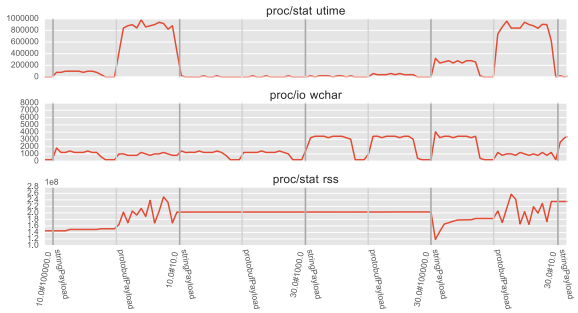

Fig. 4. Excerpt from a test for a logging component. Two test phases are executed for different parameter combinations, in this case frequency and size of events to display by the logger.

2) Data Acquisition & Recording: In order to generate and record data for later performance analysis, we apply the following approach: The test runner instantiates an external monitoring process, which obtains performance counters (or KPIs) for each of the started component processes by using the Linux `proc` filesystem, which includes aspects like CPU, memory, I/O and threads. The monitor is implemented as efficient as possible in order to minimize the load it additionally imposes on the system (e.g. below 2 % CPU usage per process on an Intel Xeon E5-1620). Acquired counters are exposed via the middleware using dedicated channels per component. In addition, the test runner exposes information about the executed test cases, phases and parameter sets via dedicated middleware messages, so that the counters can be related to this structure. For persisting the generated data, the test runner launches an instance of the middleware recording tool, in our case `rsbag`, which is configured to record the entire communication. This way, recording results can be replayed completely for detailed analysis and debugging purposes. The recording method has previously been described in Wienke *et al.* [28], including details about the performance counters. Figure 4 shows an excerpt of a test case recording for a single test case with two test phases which are executed for different parameters.

D. Test Analysis

1) Data Preparation: The output of a performance test is a file with all middleware events including the component communication, information about the test progress, and performance counters for the tested component. While this provides a good basis for detailed analyses, the file size usually prohibits to store these files for a longer time to build a database of the performance changes inside the component and random access times are slower compared to other formats. Therefore, we first transform information about the test progress and counters into a HDF5 file using the Python pandas library [29]. Here, information about the represented component revision are attached to the data, which are a human readable title (e.g. a Git hash for tests per Git commit or a time stamp for nightly builds) and a machine-sortable representation (e.g. the Git commit date or an ISO 8601 formatted date) so that executions can be ordered accordingly. In case a test has been executed multiple times for the same component revision, it is assumed that title and sort representation have the same value for all revisions (a test execution date is added to the data automatically and allows to later distinguish between test executions). These HDF5 files are the artifacts which are usually persisted for each test execution. For this purpose, a command line analysis tools was implemented, which realizes the complete analysis step. It is designed to be integrated into shell scripts, e.g. for a CI server integration.

2) Plots: As a first means of manually inspecting the performance of a component, the analysis tool allows to generate several plots from recorded data. These include the raw performance counter time series of a single execution, correlations between counters and numeric test parameters and several plots which show how performance counters have evolved with component revisions. For this purpose, counters are summarized for each test case, test phase and parameter set via mean and standard deviation and plotted for each revision of the component. This allows to track how the usage of individual counters has evolved. Figure 4 shows an excerpt from one of the generated plots.

3) Automatic Regression Detection: To detect changes in the resource consumptions of a component, we have implemented three different methods in the analysis tool. All methods take one or more test execution results (as HDF5 files) and compare the observed performance against a baseline from one or more test executions. We do not enforce a method of defining executions as baseline and test data, but at least the following modes can be found in the literature:

- "no-worse-than-before" principle [7]: the current revision is compared to the previous one (or a window of n previous revisions) to ensure that the current state is at least as good as the previous one.
- Comparison to a hand-selected baseline: all revisions can be compared to a manually selected baseline to ensure that the criteria of this baseline are met. The baseline needs to be reselected to match intended performance changes.

These modes can easily be realized using the analysis tool with different sets of HDF5 files.

Most existing methods for detecting performance regressions actually detect any change in the performance charac-

teristics of the tested system. While an automatic categorization whether a change is a degradation or and improvement in the performance would be a desirable feature, this is often not easily possible. In our own experiments we found cases where e.g. a new component revision resulted in a higher CPU usage for small workloads while the usage was improved for higher workloads. Therefore, we have implemented a mode where any change is reported and the developer has to decide (e.g. based on the plots), whether the change is acceptable.

For the actual detection of performance changes we have implemented the following methods:

- The method proposed by Foo *et al.* [19] as an example for an association rule learning based approach.
- The method proposed by Shang *et al.* [21] as a reference for a recent method based on clustering and regression.
- A basic two sample Kolmogorov-Smirnov test for each performance counter. For this purpose, the individual measurements of each performance counter across the whole test execution time (of potentially multiple executions) are assumed to form an observation and the observation from the test executions is compared to the observation from the baseline.

4) Automation: The analysis tool reports results using JUnit XML files, which can be parsed by many automation tools, e.g. the Jenkins CI server. This allows to integrate the approach with such tools which can then give feedback on potential performance regressions. Since HDF5 files from previous test executions are required to perform the analysis, we have provided a utility command which allows to use the Jenkins job artifacts feature as a lightweight database for these files. In Jenkins, a job can store build artifacts (files) as part of the execution. Jenkins persists these artifacts and makes them available via its API. The provided utility command downloads archived HDF5 files from previous test executions for use in the analysis step. This Jenkins integration is an easy way to trigger a test execution for each new software revision with automatic notification. In the future, more sophisticated tools like *LNT* [30] might be required to archive test results and make them browsable.

Generally, acquired performance results are coupled to the execution platform due to the system-specific execution times and measurements. Therefore, a dedicated host should be used for all test executions and this host should be free from other tasks to avoid resource sharing issues, which might influence the measurements. For instance, this can be realized by adding a dedicated performance testing slave to a Jenkins server. Also, dynamic frequency scaling techniques might influence the results. We therefore advise to enable the Linux `performance` CPU governor, at least for the test runtime.

V. EVALUATION

In order to validate the automatic detection of performance changes, we have implemented performance tests for several vertical components from our systems, which cover a range of different programming environments:

- `2dmap`: A Java-based visualization for person tracking results in a smart environment.
- `legdetector`: A Java-based component for detecting legs in laser scans, used by our mobile robots.
- `objectbuilder`: A C++-based component which generates stable person hypotheses from detected legs and the SLAM position of a robot.
- `logger-*`: A console-based logger for middleware events with different output styles (Common Lisp).
- `bridge`: An infrastructure component which routes parts of the middleware communication to other networks (Common Lisp).

For all of these components, tests have been written using the presented Java API and results have been processed using the aforementioned analysis methods. All tests could be generated with the provided actions which suggests that the provided set of actions is generally sufficient for writing tests for vertical components.

We have tested the presented components using the "no-worse-than-before" principle by comparing each revision against the previous one, as this is automatically possible without the necessity for a manual baseline selection. For the `2dmap`, `legdetector` and `objectbuilder` component all Git commits that could still be compiled have been used while for the `logger` and `bridge` archived component nightly builds were used.

All available analysis methods have been applied to compare their performance. Each of them requires a threshold for a numeric score to decide whether a test execution represents a performance change or not. We have used this score to compute the Area Under Curve (AUC) on ROC curves as the target metric for the classification. Since all analysis methods actually return multiple scores per test (Shang *et al.* [21] returns one score per cluster, Foo *et al.* [19] returns one score per frequent item set, and the KS-test returns one score per performance counter), these individual scores need to be combined into a single one to enable computing the AUC. We have used the min, max and mean functions for this purpose.

To get ground truth information, the generated plots displaying the evolution of performance counters across revisions have been manually examined and annotated. Additionally, the commit logs have been used, especially in cases where a decision was not easily possible from the representation. While we took great care with the annotations, we still expect some amount of errors since it is sometimes very hard to decide whether visible changes are real performance changes or caused by possible external disturbances. Especially for the nightly builds, the Git commit log was not sufficient to trace all possible changes, e.g. to the compilation environment used to create each build. These annotation issues are expected to decrease the AUC scores. Table II displays the amount of available data per component. The column "execs" indicated how often the test has been executed per revision and "changes" shows how many revisions have been manually tagged to contain performance changes.

TABLE II
AVAILABLE EVALUATION DATA PER COMPONENT

	revisions	execs	changes
2dmap	25	4	10
legdetector	14	4	4
objectbuilder	23	4	7
logger-compact	306	2	16
logger-detailed	306	2	7
logger-monitor	228	2	6
bridge	176	2	6

Based on the available data we receive the evaluation results visible in Table III. The highest scores per component are highlighted. These show that for component tests the basic Kolmogorov-Smirnov test works best. Only for some settings the method by Shang *et al.* shows comparable or slightly better scores. Especially the method proposed by Foo *et al.* does not seem to work on this kind of data.

For the results shown in Table III, the tests have been executed multiple times (cf. Table II for the actual numbers). This has been done, because several aspects of the component performance characteristics differ across runs of the same revisions. For instance, due to garbage collection timing in Java or Common Lisp programs, the memory footprint might be different across runs. Generally, we have observed that memory is usually one of the most common causes for false-positives due to such issues and averaging across multiple runs provides a way to counteract this. In contrast to component restarts during test execution (e.g. for each parameter set and test phase) this is still faster to perform due to less restarts while retaining the ability to detect performance issues like memory leaks. Additionally, effects of component initialization (warming up caches, loading files and libraries etc.) are less visible in the data. To quantify the effect of the number of test executions on the detection of performance changes, we have varied the amount of executions for all components that have been tested four times in total. Figure 5 shows the results for the most promising detection methods. While there seems to be a slight improvement for Shang *et al.* with the number of test executions, the results for the KS-test are inconclusive. On the other hand, both methods already show a reasonable performance with a single execution of the tests.

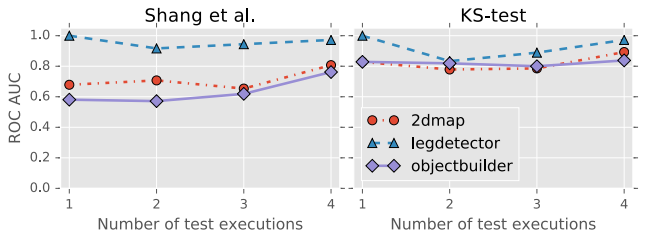

Fig. 5. Influence of the number of test executions on the two most promising detection methods. For Shang *et al.* the max aggregation method was used and for the KS-test the mean of all counter scores.

TABLE III
ROC-AUC SCORES FOR THE DIFFERENT ANALYSIS METHODS

	Foo *et al.*			Shang *et al.*			KS-test		
	min	max	ø	min	max	ø	min	max	ø
2dmap	0.50	0.72	0.71	0.81	0.81	0.83	0.50	0.50	**0.89**
legdetector	0.50	0.51	0.47	0.75	**0.97**	**0.97**	0.50	0.50	**0.97**
objectbuilder	0.50	0.63	0.66	0.28	0.76	0.55	0.50	0.50	**0.84**
logger-compact	0.50	0.49	0.51	0.45	0.56	0.48	0.59	0.59	**0.76**
logger-detailed	0.50	0.39	0.39	0.46	0.60	0.57	0.61	0.50	**0.84**
logger-monitor	0.50	0.57	0.57	0.69	**0.82**	0.80	0.48	0.50	0.72
bridge	0.50	0.59	0.59	0.55	0.56	0.48	0.44	0.49	**0.61**
ø	0.50	0.56	0.56	0.57	0.73	0.67	0.52	0.51	**0.81**

VI. DISCUSSION

We have presented a method to test individual components of robotics and intelligent systems for performance regressions introduced by code changes. The implemented framework consists of a Java API to specify tests which operate on the component's middleware interface. Special care has been taken to provide abstractions suitable for vertical components and the required data generation tasks. After test executions an analysis tool automatically decides whether a new test execution shows performance changes compared to a baseline from previous executions. By specifically supporting automation systems like the Jenkins CI server, this allows to easily construct an automated verification of performance aspects. Consequently, developers are better informed about the impact of their changes on the performance characteristics. We have verified the applicability of the concepts defined in the testing API and the effectiveness of the automatic performance change detection based on components drawn from different robotics and intelligent systems which are in active use at our labs. The idea of testing individual components for performance characteristics is a new perspective which is easier to apply than testing the whole system in the robotics context. Detected performance regression can easily be attributed to individual code units. Nevertheless, in the future, work on testing complete systems is an additional axis that needs to be performed to detect further categories of performance regressions.

The presented testing API is eventually meant only as a base tool for specifying performance tests. Java was chosen as a compromise between an acceptable coding experience and the required efficiency to generate load tests. However, the syntax is verbose and requires many code-level constructs for specifying semantic tasks. Therefore, we envision the use of a suitable DSL with embedded scripts for specifying performance tests in a more natural and readable way.

Our current implementation of the framework uses the RSB middleware as its basis. However, the concepts are generally applicable for comparable middlewares. In the testing API, the RSB-related methods are already separated and can be exchanged with other backends. A similar separation is possible also for the analysis tool.

In the future, we will also focus on further improving the detection of anomalies. One aspect commonly found in load

tests for large-scale websites are explicit warm up or tickle loads which are ignored in the final analysis and shall reduce the impact of system initialization. It is currently already possible to manually add test cases or test phases to achieve a similar behavior, but adding such a concept as a first class citizen will ensure that users of the API are aware of the issue. Despite having ignored these effects in the current evaluation, the presented detection scores already show that the system is usable. Another issue we have observed is the discretization of rarely used resources by the Linux kernel. This results in peaks in the resource usage at unpredictable times during the tests and some of the presented detection methods are sensitive to such peaks. We will improve on this in future versions. Despite the potential for improvement, the framework already provides a good foundation to detect performance degradations and has helped to identify several previously unknown regressions in our own components.

ACKNOWLEDGMENT

This work was funded as part of the Cluster of Excellence Cognitive Interaction Technology 'CITEC' (EXC 277), Bielefeld University and by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster Competition "it's OWL" (intelligent technical systems OstWestfalenLippe) and managed by the Project Management Agency Karlsruhe (PTKA).

REFERENCES

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: An open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[2] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet another robot platform," *Journal on Advanced Robotics*, vol. 3, pp. 43–48, 1 2006. DOI: 10.5772/5761.

[3] G. Steinbauer, "A survey about faults of robots used in RoboCup," in *RoboCup 2012: Robot Soccer World Cup XVI*, 2013, pp. 344–355.

[4] M. Fowler. (2006). Continuous integration, [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html (visited on 08/30/2016).

[5] M. J. Sydor, *APM best practices, Realizing application performance management*. Apress, 2011.

[6] I. Molyneaux, *The art of application performance testing, From strategy to tools*, 2nd ed. O'Reilly, 2015.

[7] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, pp. 1091–1118, 11 2015. DOI: 10.1109/TSE.2015.2445340.

[8] *Apache JMeter*, Apache Software Foundation, https://jmeter.apache.org/ (visited on 08/08/2016).

[9] *Tsung*, http://tsung.erlang-projects.org/ (visited on 08/23/2016).

[10] *Locust*, http://locust.io (visited on 08/30/2016).

[11] *NLoad*, http://www.nload.io (visited on 08/23/2016).

[12] *The grinder*, http://grinder.sourceforge.net (visited on 08/23/2016).

[13] S. Chen, D. Moreland, S. Nepal, and J. Zic, "Yet another performance testing framework," in *19th Australian Conf. on Software Engineering (ASWEC 2008)*, 2008, pp. 170–179. DOI: 10.1109/ASWEC.2008.4483205.

[14] *Gatling*, http://gatling.io (visited on 08/30/2016).

[15] M. B. Da Silveira, "Canopus: A domain-specific language for modeling performance testing," PhD thesis, Pontifical Catholic University of Rio Grande do Sul, 2016.

[16] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in *35th Int. Conf. Software Engineering (ICSE)*, 2013, pp. 1012–1021. DOI: 10.1109/ICSE.2013.6606651.

[17] T. H. Nguyen, "Using control charts for detecting and understanding performance regressions in large software," in *IEEE Fifth Int. Conf. Software Testing, Verification and Validation*, 2012, pp. 491–494. DOI: 10.1109/ICST.2012.133.

[18] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proc. 3rd ACM/SPEC Int. Conf. Performance Engineering*, 2012, p. 299. DOI: 10.1145/2188286.2188344.

[19] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *10th Int. Conf. Quality Software (QSIC)*, 2010, pp. 32–41. DOI: 10.1109/QSIC.2010.35.

[20] Ž. Žaleźničenka, "Automated detection of performance regressions in web applications using association rule mining," Master's thesis, Delft University of Technology, 2013.

[21] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proc. 6th ACM/SPEC Int. Conf. Performance Engineering*, 2015, pp. 15–26. DOI: 10.1145/2668930.2688052.

[22] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, pp. 3–22, 1 2009. DOI: 10.1016/j.jss.2008.03.066.

[23] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, pp. 634–658, 8 2010. DOI: 10.1016/j.peva.2009.07.007.

[24] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I), Reusable building blocks," *IEEE Robotics & Automation Magazine*, vol. 16, pp. 84–96, 4 2009. DOI: 10.1109/MRA.2009.934837.

[25] C. Schlegel, "Communication patterns as key towards component-based robotics," *International Journal of Advanced Robotic Systems*, vol. 3, pp. 49–54, 1 2006. DOI: 10.5772/5759.

[26] J. Wienke and S. Wrede, "A middleware for collaborative research in experimental robotics," in *IEEE/SICE Int. Symp. on System Integration (SII2011)*, 2011, pp. 1183–1190. DOI: 10.1109/SII.2011.6147617.

[27] *Protocol buffers*, Google, https://developers.google.com/protocol-buffers/ (visited on 08/17/2016).

[28] J. Wienke, S. Meyer zu Borgsen, and S. Wrede, "A data set for fault detection research on component-based robotic systems," in *Towards Autonomous Robotic Systems*, 2016, pp. 339–350. DOI: 10.1007/978-3-319-40379-3_35.

[29] W. McKinney, "Data structures for statistical computing in Python," in *Proc. 9th Python in Science Conf.*, 2010, pp. 51–56.

[30] *LNT*, http://llvm.org/docs/lnt/ (visited on 08/30/2016).