Iterative Model Driven Integration Checks of Component Based Robotic Systems

Florian Lier[1], Ingo Lütkebohle[2], and Sven Wachsmuth[1]

[1] Center of Excellence Cognitive Interaction Technology (CITEC), Universitätsstraße 21-23, 33615 Bielefeld, Germany
[2] CoR-Lab Research Institute for Cognition and Robotics, Universitätsstraße 25, 33615 Bielefeld, Germany

1 Abstract

A robot's software ecosystem often comprises a set of heterogeneous software components, acquiring, exchanging, fusioning, and deriving data to trigger a desired behaviour, state, or action of the robot. Due to the nature of component based development [1] [2], and component interaction respectively, an essential, and often crucial part of robotic system development is frequent integration testing. While unit tests already provide an established way of checking modelled constraints on single components, tests including complete component based *(sub-)*systems have not been widely studied in robotics so far. We have identified the following problem statements with respect to component based system testing so far.

1. Not all components are necessarily implemented in the same programming language, for the same operating system, or make use of exotic language bindings. Hence, the setup or bootstrapping of a sufficient test environment with an applicable configuration is often complex.
2. Previous work on component based integration testing has mainly focused on: data flow testing [3], mutation testing [4], graph-based testing [5], or has been conducted in a homogeneous programming environment [6]. Thus, previous work on integration testing has mainly focused on the very "insides" of testing a system.
3. Most of these tests are conducted manually, information about the procedure, e.g., timing, sequence (orchestration), relations, and configuration are transparent to the developer.
4. No semantic model or knowledge is explicitly given to reproduce tests.

Therefore, we propose an automated testing methodology, similar to what Bai et al. [7] have presented in the web service domain, which allows for *automated*, *frequent* and *composable* integration checks. In order to invoke automated and frequent testing, we propose the utilization of a Continuous Integration Server [8], capable of invoking well defined integration tests. Additionally, we introduce a state machine based approach to orchestrate: a) environment setup, b) system bootstrapping, c) system tests, and d) result assessment. Last but not

least, semantic knowledge about automated system tests will be compiled into an "integration test ontology". The following user-story describes the scope and purpose of our approach, which is still work in progress:

"A group of researchers develops a robotic system in a locally distributed project. The group has chosen an agile process model, allowing quick changes and adaptive planning. Diverse system components are implemented independently. Besides local instances, the testing and distribution is carried out automatically by an Continuous Integration (CI) server, accessible to all subjects of the team. For each commit, all components are individually checked via static code analysis, model checking, and unit tests. After all individual tests have been successfully performed, an integration test is automatically launched starting several subsystems, based on the artifacts previously produced by individual tests, within a predefined a environment, configuration, and constraints. The derived results of individual, and also integration tests (multiple configurations), are available to all members of the team immediately."

Our contribution to the integration test scenario described above will be based on a purposefully limited integration test ontology, which can be translated into an executable State Chart XML (SCXML) notation [9] [10], based on Harel statecharts, via Domain Specific Language (DSL). Pertaining to the integration tests mentioned above, developers are able to formalize integration tests using a DSL — based on a given test ontology — and provide it alongside with their regular code. During checkout, the DSL is translated into SCXML and executed by the CI server, invoking the system test. A trivial integration test, for instance, is the successful start of the system given a specified configuration, environment, and constraints on in- and output for the required components. In the first iteration, our work focuses on the following three fundamental problem statements: 1) The orchestration of bootstrapping/starting a system is usual dependent on the environment (Listing 1.1) and on a sequence of several component statuses, e.g., "component is successfully started if: a certain output/message is given, the process is alive, or a lock-file is present" (Listing 1.2). While these statuses, and especially their sequence, are implicitly detected by developers when starting the system manually, automated startup checks are rarely implemented natively. Due to the nature of a state machine, we implemented an initial environment setup state, exporting several environment variables used by components. Furthermore, we designed a startup state which invokes components and monitors their status, based on one (or more) of the three conditions mentioned above — continuously if required. After the startup state has been successful, the state machine switches into a testing, or logging state, followed by a result assessment state. 2) Parallelism: usually, when log or test results are to be assessed, it is convenient, and sometimes even required, to avoid huge offsets in relation to timestamps. Therefore, we also designed parallel states to allow simultaneous invocation of components or logging tools, for instance (Listing 1.3). 3) Formalism: to describe the fundamental requirements, specify test concepts, semantics, and relationships of these states and tests, we are currently investigating the basic requirements, e.g., for "system starts without failure".

Based on the findings, we will eventually compile a formal representation, in form of an ontology, to make the test domain knowledge explicit. Moreover, we find the notation in a XML-based (SCXML) manner comes with an unnecessary overhead, and is most likely prone to user-induced (syntax or logic) errors. Therefore, we will design a DSL that, on the one hand, allows for ease of use — concerning the design of an integration test, and on the other hand allows for validation against the given test ontology.

Listing 1.1. Environment Setup block

```
<data id="environment" xmlns="http://ci.clf.cit-ec.de/fsm-testing">
  <variable var="DISPLAY" val=":0.0"/>
  <variable var="PREFIX" val="/vol/clf/trunk/"/>
  <variable var="PATH" val="$PREFIX:vol/clf/bin/"/>
  <variable var="CMAKE_INSTALL_PREFIX" val="/vol/clf/"/>
  <variable var="PKG_CONFIG_PATH" val="/vol/clf/lib/pkgconfig/"/>
  <variable var="MORSE_ROOT" val="/vol/clf/bin/"/>
</data>
```

Listing 1.2. Component startup block

```
<component val="xsc2">
    <command val="xsc2_server_$prefix/etc/xsc2/config_2.cfg"/>
    <path val="$prefix/bin/"/>
    <executionHost val="localhost"/>
    <checkExecution val="True">
        <checkType val="stdout" criteria="AUTOTX_TABLE" timeout="10" />
        <checkType val="lockfile" criteria="xs2server.lock" timeout="2"/>
        <checkType val="pid" criteria="" timeout="2"/>
    </checkExecution>
</component>
```

Listing 1.3. Parallel state block

```
<state id="MOCAP_Test">
<parallel id="para">
    <state id="MOCAP_playback">
        <onentry>
            <log label="entering" expr="MOCAP_Test" />
            <fsmt:executeProgram expr="exec" value="mocap" />
        </onentry>
        <onexit>
            <log label="exiting" expr="MOCAP_Test" />
        </onexit>
    </state>
    <state id="ANGLES_Test">
        <onentry>
            <log label="entering" expr="ANGLES_Test" />
            <fsmt:executeProgram expr="exec" value="angles" />
        </onentry>
        <onexit>
            <log label="exiting" expr="ANGLES_Test" />
        </onexit>
    </state>
    <transition target="result_assessment" />
</parallel>
</state>
```

2 Acknowledgments

This work has been partially supported by the German Aerospace Center (DLR) with funds from the Federal Ministry of Economics and Technology (BMBF) due to resolution 50RA1023 of the German Bundestag. We would also like to thank Norman Köster for his enduring and excellent work on the current state of implementation as a student assistant.

References

1. D. Brugali and P. Scandurra. Component-based robotic engineering (part i) [tutorial]. *Robotics Automation Magazine, IEEE*, 16(4):84 –96, december 2009.
2. D. Brugali and A. Shakhimardanov. Component-based robotic engineering (part ii). *Robotics Automation Magazine, IEEE*, 17(1):100 –112, march 2010.
3. M.J. Harrold and M.L. Soffa. Selecting and using data for integration testing. *Software, IEEE*, 8(2):58 –65, march 1991.
4. Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649 –678, sept.-oct. 2011.
5. Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. In *Engineering of Complex Computer Systems, 2001. Proceedings. Seventh IEEE International Conference on*, pages 222 –232, 2001.
6. Ravinder Kumar and Mr Karambir Singh. A literature survey on component testing in component based software engineering. 2012.
7. Xiaoying Bai, Shufang Lee, Wei-Tek Tsai, and Yinong Chen. Ontology-based test modeling and partition testing of web services. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 465 –472, sept. 2008.
8. Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 2006.
9. Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, et al. State chart xml (scxml): State machine notation for control abstraction. *W3C Working Draft*, 2007.
10. R.S. Moura and L.A. Guedes. Simulation of industrial applications using the execution environment scxml. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 1, pages 255 –260, june 2007.