Reliable In-Field Tests for Testable Microservices of Process Control Systems

Joachim Fröhlich and Christoph Stückjürgen
Siemens AG, Munich, Germany
{froehlich.joachim, christoph.stueckjuergen}@siemens.com

Abstract

Microservices for process control systems differ from microservices for information systems. Testing of microservices is particularly crucial in industrial contexts due to the interaction of process control software and physical processes. Data probes permanently built into microservices enable reliable results from tests (probe programs) about the behavior of single and multiple microservices, even under rough and uncomfortable operating conditions in close proximity to controlled physical processes.

1 Characteristics of Microservices in a Process Control System

Regardless of the application domain, a microservice is small and focused on doing one thing well (Newman 2015). A microservice is in charge of its own data and can be deployed independently from other microservices. Functions of a microservice run as transactions entirely in an independent operating system (OS) process. Microservices for information systems are embedded in comfortable microservice containers that provide virtually unlimited resources. In contrast, microservices for process control systems (PCS) run in OS processes with little or no support from microservice containers or heavyweight components, in order to operate sufficiently close to the physical processes. PCS microservices are bound by the physics of the controlled processes and by limited operating resources. We focus on time-triggered PCS microservices because of their deterministic behavior.

2 Data Probes and Probe Points Make Microservices Testable

In-field testing must run free of side effects in order to obtain reliable statements from tests about PCS microservices. Hence, a PCS microservice must be testable by design. A testable PCS microservice contains a data probe for monitoring, manipulating (stimulating), and checking microservice data, called probe points. Data probes use exclusive time slots (CPU times), memory areas and network bandwidth for performing tests. These resources remain reserved throughout the life cycle of a microservice, independent of whether data probes actually perform tests. Figure 1 illustrates data probes and probe points in the context of a PCS microservice with process control functions, signal sensors, and signal producers (actuators).

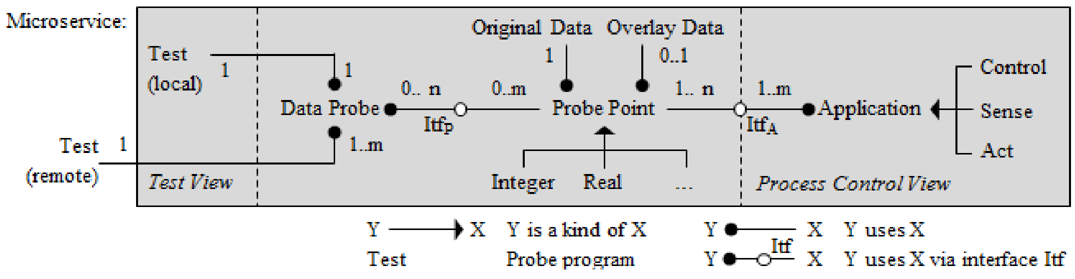


Figure 1: A PCS microservice with built-in data probes, comparable to test probes (Li et al. 2018).

Probe points can encapsulate data of different types. Data of special interest to a PCS, and hence to PCS tests, are:

- input signals from system sensors and from environment sensors
- output signals commanding system actuators
- data packets for communication between PCS microservices
- data packets for communication of the PCS with other systems
- internal state variables and parameters of PCS microservices

Application parts of PCS microservices, i.e., functions for process control, sensor functions or actuator functions (I/O drivers), do not notice that they use data encapsulated in probe points. On the application side probe points offer an interface (Figure 1, Itf_A) with operations that overload data access operators. Probe points customize the implementation of data access operators so that applications, such as machine control functions (Figure 1, Control) and signal sending functions (Figure 1, Act), either work with the encapsulated original data or with the encapsulated overlay data. On the data probe side probe points offer an interface (Figure 1, Itf_P) with special-purpose operations for reading (monitoring) and manipulating encapsulated data. For example, when injecting faults into signal variables or when adjusting the state of a PCS process, a data probe writes overlay data. In this way a data probe manipulates probe points for testing purposes. An application accessing a probe point always writes original data but reads either original data or, if manipulated, overlay data, without noticing the manipulation. When checking microservice logic a data probe reads and checks original data.

3 System of Microservices controlling a Mobile Robot

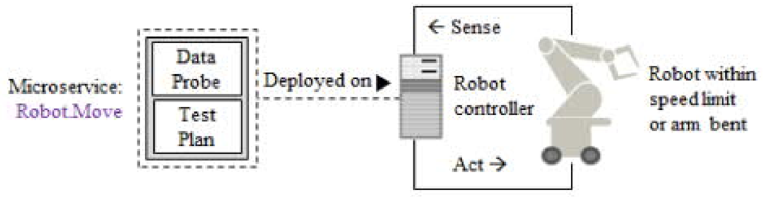
The system under test is a mobile robot with an arm, gripper and workbench. Basically, the mobile robot can fetch and carry loads. The control logic of the robot is implemented as a system of microservices. For example, the microservice *Robot.Move* controls the motion of the mobile robot. Another microservice, *Robot.Grip*, controls the robot's gripper. The robot can work in groups with other (mobile and stationary) robots and with humans. Hence, the mobile robot must be reliable and safe. A safe mobile robot must never extend its arm (manipulator with gripper) beyond its base corpus when moving faster than a given safe speed limit (Machin et al., 2014). This requirement can be formulated as a safety invariant.

3.1 Field Test of a Safety Invariant in Scope of One Microservice

The data probe built into the microservice Robot.Move checks whether the mobile robot moves safely according to this safety invariant. To this end the data probe performs an appropriate test plan formulated as constraint on 2 process variables that *Robot.Move* implements as probe points (Figure 2).

```
01: PROBE Mobile robot moves
02: WITH v0 // speed limit
03: TRIGGER TRUE
04: CYCLE 0 TO $ DO
05:   Robot.Move.v < v0
06:   OR Robot.Move.a
07: STOP ON FAIL
```

(a)



(b)

Figure 2: Data probe executes test plan (a) while the containing microservice controls robot moves (b).

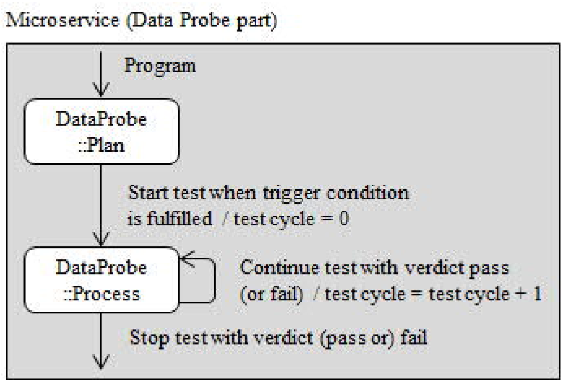
The robot's ground speed (process variable *v*) must be less than a given safe speed limit *v0* (Figure 2.a, line 05). Otherwise the robot's arm must be bent (Figure 2.a, line 06). Process variable *a* indicates whether the arm is bent (*a* is true) or extended (*a* is false).

The test is executed within the time slot of a system cycle exclusively reserved for the data probe, starts immediately with the robot's operation (Figure 2.a, line 03) and runs from the first system cycle to the last system cycle without break (Figure 2.a, line 04), if the microservice functions as expected. Since, in this case, the test shall check whether the safety invariant holds in each cycle during an entire mission, the system cycle counter and the test cycle counter have the same value. The test shall stop (Figure 2.a, line 07), and hence the test cycle counter as well, if the assert (Figure 2.a, lines 05-06) fails. The microservice continues to operate.

The test plan is written in ALFHA (Fröhlich and Stückjürgen, 2017). Originally, ALFHA has been designed for specifying tests of time-triggered, distributed, fault-tolerant process control systems (Frtunikj et al., 2015). The C++ implementation of the ALFHA test plan reveals some details of the operating procedure of the test and the data probe (Figure 3).

```
void DataProbe::Plan(void) {
   this->Name("Mobile robot moves");
   Monitor m;
   m.Parameter("v0").Variable("v").Variable("a");
   m.TriggerIf(true);
   m.TestCycleFrom(0);
   this->InTestScope(m);
}
bool DataProbe::Process(const Monitor& m) {
   if (m["v"] < m["v0"]) return Continue(pass);
   if (m["a"]) return Continue(pass);
   return Stop(fail);
}
```

(a)


Microservice (Data Probe part)
Program
DataProbe ::Plan
Start test when trigger condition is fulfilled / test cycle = 0
DataProbe ::Process
Continue test with verdict pass (or fail) / test cycle = test cycle + 1
Stop test with verdict (pass or) fail

(b)

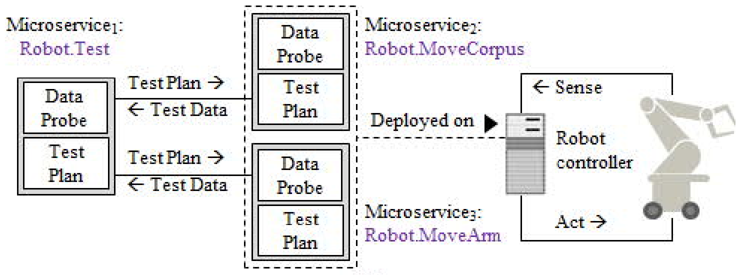
Figure 3: Test plan in C++ (a) and the basic flow of control when processing the test plan (b).

3.2 Field Test of a Safety Invariant in Scope of Two Microservices

In a different configuration, the functions that control robot motions are implemented in two microservices: One microservice controls the motion of the corpus and is primarily responsible for process variable ground speed (*v*). The second microservice controls the motion of the arm and is primarily responsible the process variable that indicates the arm state (*a*). The test setup and scope for checking the safety invariant of robot motions as stated above is adapted accordingly (Figure 4).

```
01: PROBE Mobile robot moves
02: WITH v0
03: TRIGGER TRUE
04: CYCLE 0 TO $ DO
05:    Robot.MoveCorpus.v < v0
06:    OR Robot.MoveArm.a
07: STOP ON FAIL
```

(a)



(b)

Figure 4: Test plan (a) is divided into 3 subplans for 3 data probes that execute them during robot motion (b).

Testing the safety invariant now means testing a distributed system, although it is a simple system comprising only the two microservices *Robot.MoveCorpus* and *Robot.MoveArm*. On a high level, the only change is the qualification of process variables (probe points) with the names of the responsible microservices (compare Figure 4, lines 05-06, with Figure 2, lines 05-06). This test is executed by a data probe inside a microservice, *Robot.Test*, specialized in performing microservice integration tests. The test microservice is the pivotal, independent body that splits the test plan into subplans to be executed by the data probes running inside the microservices under test. On a low level, the C++ implementation and its allocation among the three involved microservices reveals some details of the operating procedure (Figure 5).

```
// Figure 4, microservice1 Robot.Test
void DataProbe::Plan(void) {
   this->Name("Mobile robot moves");
   Monitor m;
   m.Microservice("Robot.Test").Parameter("v0");
   m.Microservice("Robot.MoveCorpus").Variable("v");
   m.Microservice("Robot.MoveArm").Variable("a");
   m.TestCycleFrom(0);
   m.TriggerIf(true);
   this->InTestScope(m); //Implies subplan distribution
}

bool DataProbe::Process(Monitor& m) {
   if (m["Robot.MoveCorpus"]["v"] < m["Robot.Test"]["v0"])
      return Continue(pass);
   if (m["Robot.MoveArm"]["a"]) return Continue(pass);
   return Stop(fail);
}
```

(a)

```
// Figure 4, microservice2 Robot.MoveCorpus
void DataProbe::Plan(Monitor& m) {
   this->InTestScope(m); //Monitor "v"
}

bool DataProbe::Process(Monitor& m) {
   this->Forward(m); // Async. to Robot.Test
   return Continue(pass);
}
```

(b)

```
// Figure 4, microservice3 Robot.MoveArm
void DataProbe::Plan(Monitor& m) {
   this->InTestScope(m); //Monitor "a"
}

bool DataProbe::Process(Monitor& m) {
   this->Forward(m); // Async. to Robot.Test
   return Continue(pass);
}
```

(c)

Figure 5: Test plan (a) for the test microservice and subplans (b), (c) for the microservices under test.

4 Summary and Conclusion

Reliability and effectiveness of tests of microservices in the context of a process control system depend on several factors, such as on the test tasks to be performed (monitor, manipulate, relate and check process variables); on the location, number and structure of process variables in test scope; on the capability of the test to act correctly in time; on the availability of test resources reserved for performing field tests; and on the expressiveness of the test language. Testable microservices with built-in data probes and probe points address all these points. Data probes are controlled by test plans specified in ALFHA and implemented in C++. Data probes and related concepts can be used to monitor and check the control logic of a robot that is implemented as a system of time-triggered microservices. It seems that microservices of any application domain and implementation language can profit from tests with built-in data probes. Data probes, test programs (plans) for data probes and probe points can be implemented with rather simple, basic object-oriented programming techniques as the C++ code snippets indicate. Data probes and related concepts contribute to solving the observability problem faced by tests of distributed real-time systems (Schütz, 1994).

References

Fröhlich, J., Stückjürgen, C. (2017). Verification of an Autonomous System at Runtime with Built-in Test Probes. In *Proceedings of the 27th International Symposium on Software Reliability Engineering Workshops* (pp. 9-11). IEEE, October 23-26, Toulouse, France.

Frtunikj, J., Fröhlich, J., Rohlfs, T., Knoll, A. (2015). Qualitative Evaluation of Fault Hypotheses with Non-Intrusive Fault Injection. In *Proceedings of the 5th International Workshop on Software Certification* (pp. 160-167). IEEE, November 2-5, Gaithersburg, MD, USA.

Li, F., Fröhlich, J., Schall, D., Lachenmayr, M., Stückjürgen, C., Meixner, S., Buschmann, F. (2018). Microservice Patterns for the Life Cycle of Industrial Edge Software. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. ACM, July 4–8, Irsee, Germany.

Machin, M., Dufossé, F., Blanquart, J.-P., Guiochet, J., Powell, D., Waeselynck, H. (2014). Specifying Safety Monitors for Autonomous Systems Using Model-Checking. In *Proceedings of the 33rd International Conference on Computer Safety, Reliability, and Security* (pp. 262-277). Springer LNCS 8666, September 10-12, Florence, Italy.

Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly.

Schütz, W. (1994). Fundamental Issues in Testing Distributed Real-Time Systems. In *Real-Time Systems* (pp. 129-157). Volume 7, Issue 2, Springer.