EasyChair Preprint

№ 5390

# Designing and Verifying Microservices Using CSP

Jeremy Martin



April 25, 2021

# Designing and Verifying Microservices Using CSP

Jeremy M. R. Martin
School of Computing
University of Buckingham
MK18 1EG
United Kingdom
computing@buckingham.ac.uk

***Abstract*— Microservices Architecture is a popular pattern used for building complex IT systems in an incremental, sustainable, and scalable fashion. However, compared with traditional monolithic solution architectures, it introduces a higher degree of concurrency which might result in subtle bugs arising, such as race conditions, deadlocks, and lack of data consistency. I shall illustrate this using a worked example of an automated insurance claims payment service which exhibits a bug whereby a particular claim might be settled twice. I shall use the CSP formal modelling language and the FDR refinement checker to prove some results about this bug and how to fix it.**

***Keywords*— *Microservices, Concurrency, Race Conditions, Model Checking, CSP.***

## I. Introduction

Microservices architecture [1] is a modern flavour of Communicating Sequential Processes [2, 3], based on fine-grained web services and lightweight communication protocols. It represents a fundamental shift in solution delivery practice away from building complex, multi-tiered monoliths. Its main principles are as follows:

1. An IT system is delivered as a set of loosely coupled web services.
2. Each service implements a specific business capability.
3. Services can be developed independently from each other, potentially by different teams using different programming languages.
4. Communication between services uses technology agnostic protocols [3].
5. Microservices are small, so that they are suitable for a 'continuous delivery' engineering approach.

New business requirements can be addressed by making localised changes to individual self-contained microservices, offering cost-effectiveness and agility.

However, since this approach typically produces a highly concurrent system, dangers lurk such as deadlocks and race-conditions.

I shall illustrate this with a worked example of an automated insurance claims payment system which may exhibit a bug where a claim can potentially be paid out twice. I shall use the CSP (Communicating Sequential Processes) formal modelling language to prove some results about this bug and how to fix it.

## II. Case Study: Automated Insurance Claims Payment System

Let us now consider a microservices implementation of an automated insurance claims payment system. This is based on a real prototype system that I recently helped to develop for the London Insurance Market.

The system incorporates an element of artificial intelligence which could result in a claim being paid before the policy owner has even become aware of having suffered a loss. For example, consider a wildfire being reported in a region of California where an insurance company provides property cover. The payment system would be alerted of the nature and location of the event. It would then automatically check for any properties in the vicinity for which it provides insurance cover. For each such property it would then request an intelligence service to estimate the level of loss automatically, by analysing satellite or drone images. Finally, the payment engine could decide to pay out on certain losses, even in the absence of any claim, to reduce potential legal or administrative costs.

The claims system comprises four microservices: an 'alerting' service, a 'policy' service, an 'intelligence' service and a 'payment' service. Figure 1 illustrates the communication architecture: showing how the services invoke each other's interface functions.

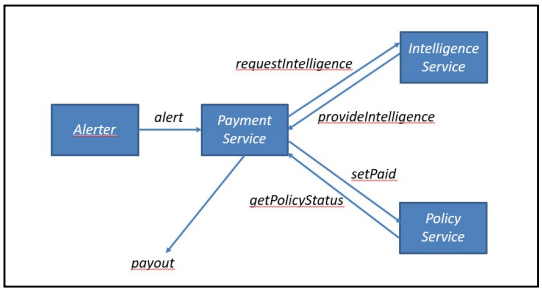


*Figure 1. Communication Diagram for Claims Payment System*

## III. CSP Analysis

In this section I shall be using the $CSP_M$ language to provide a formal definition of this microservices design and the FDR model-checking tool to establish whether it satisfies the following requirement: that the system can never pay out twice on the same claim.

$CSP_M$ is a lazy functional language with built-in support for defining CSP processes. It also allows assertions to be made about the resulting CSP processes. FDR (Failures

Divergences Refinement) is an automated checker which validates assertions about processes, defined using $CSP_M$. Rather than providing a general tutorial for $CSP_M$, I shall explain the various constructs as they are introduced in the process definitions below.

A $CSP_M$ program defines a collection of stateless processes which communicate messages with each other over shared channels. Let us start by defining some data types and channels for the messages that are transmitted in our insurance claims payment system when one of the microservices makes a call to a function of another microservice.

**datatype** *policy = Pol1 | Pol2*
**datatype** *damage = Unknown | None | Partial | Total*
**datatype** *stat = Paid | Unpaid*

The datatypes are deliberately defined in an abstract way, with just enough values to enable us to prove something useful about the system in question. (See reference [3] Section 15.2 for further details.) So, for instance, we define the 'policy' datatype to have only two allowed values. The main reason for doing this is because the FDR tool works by performing an exhaustive search of all possible states of the system being checked, so it is desirable to keep the size of the search space as small as possible so that checks can be performed in a reasonable amount of time.

**channel** *payout, requestIntelligence, setPaid*: *policy*
**channel** *provideIntelligence*: *policy.damage*
**channel** *getPolicyStatus*: *policy.stat*
**channel** *alert*

Some of the channels have an aggregate datatype, e.g. channel 'getPolicyStatus' has type 'policy.damage'. A single communication event on that channel contains both a policy value and a damage status.

The prototype system that I recently worked on was found, during testing, to exhibit a bug where a claim might be paid twice if a message became stuck in a particular channel. I shall now present a series of potential designs and use the FDR tool to determine whether they are susceptible to this problem. I aim to show how useful the CSP approach can be for diagnosing and fixing problems like this at design time, avoiding complex remediation issues post-implementation.

### A. Initial Design

Let us start with an obviously bad design which is guaranteed to fail. This very unrealistic example is included only for the purpose of illustrating how the overall approach works – of course we would never design a real system this way.

*IMPLEMENTATION1 = payout.Pol1 → payout.Pol1 →* ***STOP***

Every process in $CSP_M$ is defined using an algebraic expression since the CSP language obeys mathematical laws. Here we are introducing a special predefined process, called '***STOP***', which represents a blocked process unable to do anything. And we are also using the prefix operator → which connects an event to a process. This means that *IMPLEMENTATION1* is a process which transmits a message *Pol1* twice on channel *payout* and then stops and does nothing else. In other words, it pays out twice on policy 1 for no reason at all!

Now we will introduce a specification process that expresses the desirable behaviour that the payment system will pay out on policy 1 either once or not at all before it stops.

*SPEC =* ***STOP*** ⊓ *payout.Pol1 →* ***STOP***

A CSP operator which represents an *internal choice* between two different processes, written ⊓, has now been introduced. Here the specification means that the payment system makes some internal decision as to whether to pay the claim exactly once or not at all.

To check whether our bad implementation meets this specification we ask FDR to check a refinement assertion as follows.

**assert** *SPEC* $\sqsubseteq_{FD}$ *IMPLEMENTATION1*

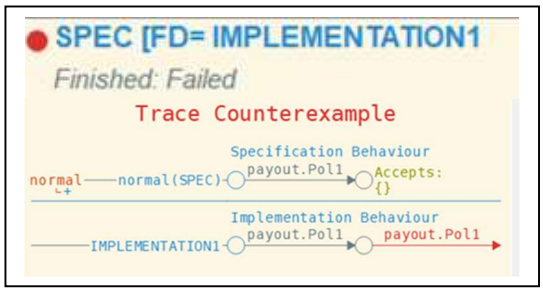


*Figure 2. Counter example found for the initial design.*

FDR proceeds to check all the behaviours of both systems to see whether the implementation exhibits any behaviour that is not possible for the specification, and it rapidly finds a counter example, as we would expect.

### B. A More Realistic Design.

Now we shall create a more realistic design, yet still somewhat abstracted. This will be as a collection of communicating microprocesses, intelligently processing the claim, corresponding to the system described in section II.

We first model the alerting process as sending out a sequence of four alert signals and then stopping.

*ALERTER = alert → alert → alert → alert →* ***STOP***

Next we model the intelligence service as waiting to receive a request for information about a potential claim related to policy, making an internal decision, and then replying with an assessment of the loss: none, partial, total or unknown. After which it returns to its initial state, as indicated by the recursive syntax.

*INTELLIGENCESYSTEM = requestIntelligence?x → (*
*provideIntelligence.x.Unknown→INTELLIGENCESYSTEM*
*⊓ provideIntelligence.x.None →INTELLIGENCESYSTEM*
*⊓ provideIntelligence.x.Partial → INTELLIGENCESYSTEM*
*⊓ provideIntelligence.x.Total→INTELLIGENCESYSTEM*
*)*

Note that we have introduced some extra syntax here to represent message input. A process *chan?x→P(x)* waits to receive a message *x* from another process along channel *chan* and then proceeds to execute process *P(x)*.

The payment system waits to be triggered by a message from the alerting service, and then checks each policy in turn using a subprocess which requests information from the intelligence service and pays the claim only if two conditions hold: that the claim has not already been paid and that the loss has been estimated as total by the intelligence service.

```
PAYMENTSYSTEM =
alert → CHECK(Pol1); CHECK(Pol2); PAYMENTSYSTEM

CHECK(pol) =
requestIntelligence.pol → provideIntelligence!pol?r →
getPolicyStatus.pol?s →
if r == Total and s == Unpaid then
   payout.pol → setPaid.pol → SKIP
else
  SKIP
```

We have now introduced another CSP operation for the sequential composition of two processes, written ';'. We have also used the '**if... then... else**' clause to specify conditional behaviour depending on the values of the process parameters.

'***SKIP***' is another special process representing clean termination. '***SKIP***' is subtly different from '***STOP***' because it may be extended by sequential composition with another process, whereas '***STOP***' is permanently blocked [7].

The policy service is implemented by simple 'getter' and 'setter' functions provided for each policy, making use of the *external choice* operation, written □, which allows an outside process to control which option is selected.

```
POLICY(x, y) =
(getPolicyStatus.x.y →POLICY(x, y)) □
(setPaid.x → POLICY(x, Paid))
```

We need to bring these processes together using parallel operators, which combine two process to create a new one.

```
IMPLEMENTATION2 =
(POLICY(Pol1, Unpaid) ||| POLICY(Pol2, Unpaid) |||
INTELLIGENCESYSTEM ||| ALERTER)
[| {|requestIntelligence, alert, getPolicyStatus,
    provideIntelligence, setPaid|} |]
PAYMENTSYSTEM
```

We are using two different types of parallel operator here. The first type is called 'interleaving' and is written '|||'. This means parallel composition with no communication – each constituent process is free to proceed without hindrance from the other. We use interleaving to combine all the independent satellite processes of the payment system, as shown in the communication diagram of figure 1. (Note that two copies of the policy process are included – one for each policy in our highly abstracted representation.)

The second parallel operator is called 'generalised parallel' and is written '[| {|*channel1, channel2, …*|} |]'. It specifies a list of channels on which both processes must synchronise to form the new process. We use generalised parallel to combine the payment process with the subnetwork of satellite processes described above.

Henceforth, when framing assertions, we shall apply the 'hiding' operator, written '\ {|*channel1, channel2, …*|}', to the implementation process in order to conceal every event except *payout.Pol1*. This is so that we can examine its behavior regarding paying out claims against a specific policy and check whether it could ever pay the claim twice.

```
assert SPEC ⊆FD
IMPLEMENTATION2 \
{|requestIntelligence, alert, getPolicyStatus,
  provideIntelligence, setPaid, payout.Pol2|}
```


*Figure 3. Second design meets specification*

On this occasion FDR reveals that the assertion has passed, and a claim will not be paid twice.

### C. Introducing a Message Bus

In fact, the system that I implemented for the London Insurance Market had another component – a message bus. Whenever a claim was paid the policy status was updated asynchronously using the bus, with the aim of helping to improve claims processing efficiency.

Let us now refine the abstract CSP design to incorporate the message bus. The new process definitions are as follows.

```
CHECK'(pol) =
requestIntelligence.pol→provideIntelligence!pol?r→
getPolicyStatus.pol?s →
if r == Total and s == Unpaid then
  payout.pol → SKIP
else
  SKIP

MESSAGEBUS = payout?pol → setPaid.pol → MESSAGEBUS

PAYMENTSYSTEM' =
alert → CHECK'(Pol1); CHECK'(Pol2); PAYMENTSYSTEM'

IMPLEMENTATION3 =
(POLICY(Pol1, Unpaid) ||| POLICY(Pol2, Unpaid) |||
INTELLIGENCESYSTEM ||| ALERTER)
[|{|requestIntelligence, alert, getPolicyStatus,
    provideIntelligence, setPaid|}|]
(PAYMENTSYSTEM' [| {|payout|} |] MESSAGEBUS)
```

**assert** *SPEC* $\subseteq_{FD}$
*IMPLEMENTATION3* \
{|*requestIntelligence, alert, getPolicyStatus,*
*provideIntelligence, setPaid, payout.Pol2*|}

Following this modification, FDR now reports that the specification fails. It provides evidence for this in the form of

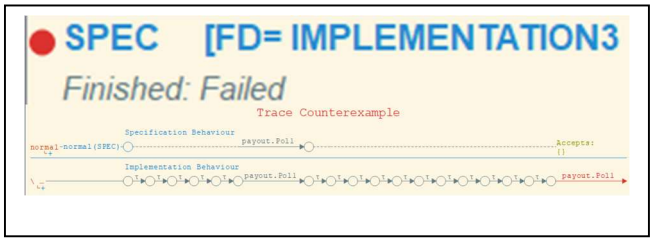


*Figure 4. Introducing message bus introduces bug*

the shortest sequence of events that the system can perform which would trigger this problem. Introducing the message bus has resulted in a bug where a claim might be paid out twice if a message gets stuck on the bus and, in the meantime, the alerting service sends out a second alert.

I observed this bug in the prototype system that I worked on, which was the trigger for commencing this analysis. It was very useful to determine how a well-meaning attempt to improve efficiency using a seemingly small modification to a microservices application could introduce such a serious defect. Taking out the message bus resolved the bug.

## IV. Conclusions

For all its benefits, a significant complication of the microservices architecture is that it introduces additional concurrency into IT systems which could lead to unexpected problems if not sufficiently well understood.

By using a formal modelling language and proof tool, we can detect such problems at the design stage and avoid having to fix expensive mistakes later.

I have shown how the CSP language and associated refinement tool, FDR, may be used in this way, by working through the specific example of an automated insurance claims payment system.

The model-checking approach necessitated creating simplified abstract datatypes and processes compared with the actual system being designed, in order to keep the size of the state space being checked manageable. Using this approach effectively requires being able to judge how far one might go with abstraction: having just enough data values and process instances to enable us to prove something useful about the system in question.

## References

[1] Sam Newman, “Building Microservices: Designing Fine-Grained Systems,” O’Reilly, ISBN: 978149195 0357, 2014.

[2] C.A.R. Hoare, “Communicating Sequential Processes,” Prentice-Hall, 1985.

[3] A.W. Roscoe, “The Theory and Practice of Concurrency,” Prentice-Hall, 1998.

[4] Leonard Richardson and Sam Ruby, “RESTful Web Services,” O’Reilly, 2007.

[5] T. Gibson-Robinson, P. Armstrong, A, Boulgakov and A. W. Roscoe. “FDR3: a parallel refinement checker for CSP,” Int J Softw Tools Technol Transfer 18, 149–167 2016.

[6] J.B. Scattergood, “Tools for CSP and Timed CSP,” Oxford University D.Phil thesis, 1998.

[7] Thomas Gibson-Robinson and Michael Goldsmith, “The Meaning and Implementation of SKIP in CSP”, Proceedings of Communicating Process Architectures 2013, ISBN 978-0-9565409-7-3. 2013.