ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC001

# USING BDD TESTING IN SKAO: CHALLENGES AND OPPORTUNITIES

V. L. Allan, University of Cambridge, Cambridge, UK*
G. Brajnik, University of Udine and IDS, Udine, Italy
R. Brederode, SKAO, Macclesfield, UK
on behalf of the The SKA Software Collaboration

*Abstract*

The SKAO (Square Kilometre Array Observatory) is one observatory, with two telescopes on three continents. It will be the world's largest radio telescope o, and will be able to observe the sky with unprecedented sensitivity and resolution. The SKAO software and computing systems will largely be responsible for orchestrating the observatory and associated telescopes, and processing the science data, before data products are distributed to regional science centres. The Scaled Agile Framework (SAFe™) is being leveraged to coordinate over thirty lean agile development teams that are distributed throughout the world. In this paper, we report on our experience in using the Scaled Agile Framework, the successes we have enjoyed, as well as the impediments and challenges that have stood in our way.

## INTRODUCTION

In this paper, we will provide an account of our attempts to adopt Behaviour Driven Development (BDD) and system testing, particularly for our control system based on Tango, with the goal of providing testers of control systems for other instruments with enough information to decide whether to use such an approach themselves. We will briefly provide the context of the SKAO (SKA Observatory), then look at our testing goals and challenges, focusing on automated testing. We will then explain what BDD testing is and what it has to offer us, and the progress we have made towards our goals while trying to use the approach. We look at the challenges imposed for creating testware to implement BDD tests for finite state automata. We will also discuss the issues we have experienced with roll-out, particularly in the context of the control system, and our current plans. We explore the organisational structures that hinder and help us, documenting our experience and conclusions for the benefit of future decision makers.

## THE SKA PROJECT

The Square Kilometre Array Observatory is one observatory, running two telescopes, over three continents. The headquarters are in the UK, and there are two telescopes: one is a low frequency (50-350MHz) telescope, consisting of hundred of thousands dipole antennas grouped together into stations (referred to as the Low telescope) in the desert in Western Australia, and the other is a mid frequency telescope (350MHz-15.4GHz) consisting of hundreds of dishes (the Mid telescope) in the Karoo desert in South Africa [1,2]. To build these telescopes, a global collaboration spanning multiple countries and timezones has been established. Rees provides a more detailed account of the current state of the project [3].

In Fig. 1, we can see the two telescopes (the Low Array and Dish sub-systems on the diagram) are each connected to a Central Signal Processor (CSP) and a Science Data Processor (SDP), plus numerous supporting sub-systems (Synchronisation and Timing, Network Manager, Telescope Manager Control (TMC), a High Performance Compute Platform, plus eventually a Very Long Baseline Interferometry (VLBI) sub-system and SRCs (SKA Regional Centres)). These systems are used to collect astronomical signals from the sky (using the antennas and dishes in the desert), correlate those signals in the CSP for each telescope, then process those data on supercomputers, turning them into a product that can be delivered to SRCs for use by scientists. Both telescopes are controlled using the Tango control system, so most sub-systems will have one or more Tango devices which allow the transfer of commands and monitoring data, including system health data [4].

Each of these sub-systems is made up of multiple components. For example, the TMC contains a Central Node, which interfaces with the tools for defining telescope observations, a Subarray node, which controls a subset of the dishes or antennas for the relevant telescope, plus components to control and monitor the CSP, SDP, and dishes/antennas for each telescope.

To organize the software development work to create these components, we use the Scaled Agile Framework®(SAFe®), which allows us to co-ordinate multiple development teams on a common cadence [5]. Our teams are grouped into Agile Release Trains (ARTs), which deliver the telescope software. These ARTs span several countries; indeed, some teams are also highly distributed.

The work is cadenced using PIs (Planning Intervals) that last one quarter; each of these begins with a review of the previous PI, and then there is a Planning week where we converge on our plans for the subsequent PI, based on goals that are set by software architects and product managers (PMs). While we can pivot during a PI, that is generally not desirable; therefore major changes to or evolution of our testing process will usually occur not more than once per quarter.

## TESTING GOALS AND CHALLENGES

Brajnik *et al.* have noted that the SKAO has several goals and challenges when considering software testing [6]. Our

* vla22@cam.ac.uk

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-M04BC001



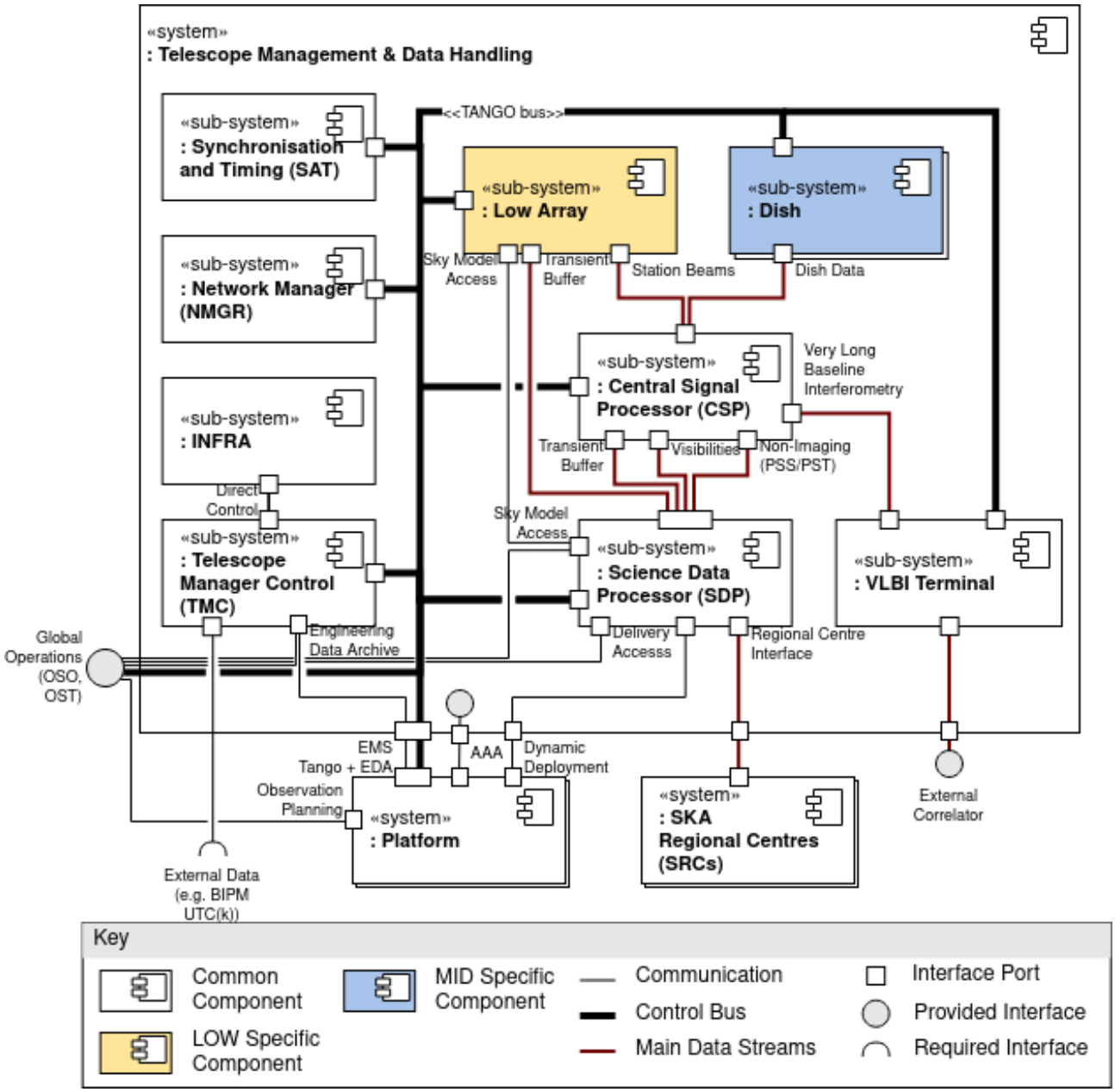


Figure 1: Component and Connector diagram of the top level of the SKAO software system.

overriding goal is to have a sustainable process, so that developers and testers do not suffer burnout and the process continuously yields a return on investment. While we will use manual tests, especially for user acceptance testing, we also want to have as many tests automated as possible. The telescope has a planned 50 year lifespan, so we anticipate many changes to the software, whether this is to the operator Graphical User Interfaces (GUIs), the underlying compute platform software, operating systems, or to other software libraries on which we depend. We have a goal to enable developers to create new releases in less than one working day; this is not feasible without extensive automated software testing [7]. Thus to make the process sustainable, it is important to automate as much as possible.

In practice this means:

- the testing process needs to be performed quickly and frequently, so that there are no disincentives to perform the tests before deployment,
- the process should support the development team, by helping them identify and localise bugs quickly and easily,
- the process should support all stakeholders by helping them understand and validate requirements and specifications, and
- the process is economically viable.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC001

### *Our Challenges*

- Our software covers many different domains, including, but not limited to: high performance computing, control systems, signal processing, radio astronomy specialised applications, and GUIs. This often requires different test approaches to be used for the different domains.
- Diverse levels of skill of developers, architects,and test engineers, make it difficult to develop a unified and consistent language and to share effectively problems and practices.
- High levels of team autonomy, as teams have a good deal of latitude in their detailed implementation, as well as in the specification and APIs of components. This is great for trying out new approaches and for using the high specialised competences located within specific teams, but it also can lead to unsupportable diversity of tools and approaches; hence co-ordinating around the best approach is difficult.
- Testware and integration/system test development needs co-ordinating across multiple teams, and this can lead to ownership problems and alignment problems.
- Some of our tests require specialised hardware (e.g. Field Programmable Gate Arrays (FPGAs)) and hardware-in-the-loop, which complicates the infrastructure and management of test environments, as explained in .
- Our integration testing is under-resourced, as a gap arose as we transitioned from the design phase to the construction phase, where some responsibilities for testing were covered, but not all of them.
- Our highly distributed nature, with developers from many cultures, and many widely-separated timezones.

Most of these challenges are not unique to SKAO. There are many highly distributed organisations, or projects where the developers are not co-located, including open source software projects and multinational companies. There are many organisations that are working with specialist hardware. Most organisations have to contend with diverse skill levels, if only because of the need to recruit new people. However, these challenges combine; because of our distributed teams, it is hard to get the people with the expertise together, as they can be in timezones 4-8 hours apart, and the varying skill levels may mean that teams can get blocked by not having the knowledge to proceed. Because of the particular timezone configuration, some timeslots are highly contested because of the large number of teams needing to communicate, reducing the opportunity for ad-hoc meetings. Similarly, the high level of team autonomy means that some teams have developed good solutions, but we then have the challenge of sharing that solution, or adapting that solution so that it is more generally applicable. However, we have high levels of support from senior management, who are keen to build in quality, so there is support for testing and integration activities.

### *Implementation*

To implement these goals we have:

- set up a Testing Community of Practice. This is something that SAFe recommends in order to cut across potential silos caused by the grouping of teams into ARTs, and these are designed to foster best practice, and share knowledge [8].
- created a strategy and policy document, that we routinely update (we have an update planned for the current PI) [9].
- engaged with the PI planning process each quarter to set out more detailed plans. The success (or otherwise) of the specific approaches tried recently are outlined in

## OUR SOFTWARE TESTING PROCESSES

ISTQB defines component integration testing as focusing on the "interfaces and interactions between components", and notes that it is heavily dependent on the testing strategy adopted [10].

They go on to define System testing as focusing on the "overall behaviour and capabilities of an entire system or product", including functional and non-functional testing of quality attributes, and define System integration testing as focusing on the interfaces of the System Under Test (SUT) to other services and/or external environments [10].

When testing a complex system with many moving parts, there are two major things one can concentrate on: one is interfaces, where contract testing techniques are useful; the other is the behaviour of the system as a whole. While we have investigated PACT and contract-based testing [11], we have had more success pursuing the behavioural approach, using BDD [12].

BDD is a development approach, which starts from the goals of the organisation. It then uses scenarios and specification by example, which results in a specification for a feature, focused on the most relevant examples [13]. The scenario development helps foster a common language to describe the system; the examples help support a test-first approach, and also support the development of living documentation [14].

The starting point is to formulate these scenarios that describe the behaviour of the SUT, using the gherkin syntax. This syntax consists of four words: Given, When, Then, And. This allows one to set up complex scenarios, using an informal English-based language that can, and should, refere to the identified domain specific language (DSL). These scenarios can be used for either or both of manual or automated testing.

The Given keyword starts a sentence that says what condition or state you expect your system to be in when starting a

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC001



test. This condition can then be set up using the features of the underlying test automation framework. Multiple starting conditions (perhaps for other sub-systems) can be defined by using the And keyword. The When keyword starts a sentence that defines a trigger: this can be a change of state in the system (for example, a failure) or a deliberate action by a user (e.g. sending a command). Finally, the Then step states what should happen as a result of the Then step. Again, multiple results can be chained together using the And keyword. This is exemplified below:

```
Given I connect to an SDP subarray
And obsState is READY
When I call Scan
Then obsState is SCANNING
And scanID has the expected value
```
[15]

Best practice is that there should be a single action or change that is triggered when performing the When step. This makes it easier to reuse the step, and to isolate what went wrong if the test fails. Complex series of commands can be put into a Then step; however, those should be built on top of, and run subsequent to, the more isolated Then steps. Tables of examples can be added to parameterise the test, to explore more of the test space. This allows for running complex scenarios, but after ensuring the basic behaviour of the system is solid.

The great utility of BDD tests is in formulating new features. BDD tests are specified by example; this works well with defining automated tests, and also helps developers assess and code the boundaries of a new feature. These discussions help bridge the gap between the conception that the architects and designers, the developers writing the code, and the testers and users who will test and use the system, have about system behaviour.

This thus provides living documentation of the system: we have a set of examples to guide our intuition about how the system should behave, that are then tied to the code, and if we wish to change the behaviour of the system, our tests will fail as the system is updated, thus causing us to update our descriptions to reflect the new behaviour. Similarly, if the intended behaviour of the system is not changing, but some of the system software is (whether our own code, updated libraries, new operating systems), we can detect regressions.

### *SKAO Testing Environments and Process*

In SKAO, we start from a position where we expect (and nearly always have) >80% source code coverage achieved by unit tests for repositories containing individual components. Our integration test process encourages testing those components with mocks or emulators to perform component integration testing. Some of those components may be within the same sub-system; others may be in a different sub-system. The integration process then encourages tests of sub-systems with mocked/emulated external interfaces in our various test environments, described below.

The final integration can only be performed with the production system, where we can get a complete signal chain for multiple dishes or stations, supported by the full compute hardware provision. This requires a rather complex integration environments, as the costs of replicating a full production system outweigh the benefits. However, we are using a staged roll-out plan, so the first production system (due in 2024) will act as a prototype and testbed for the larger-scale systems that will follow.

We have three major environments, PSIs (prototype system integration environments), intended to trial hardware components of the signal chain, ITFs (integration test facilities), where hardware and software is qualified before deployment to the telescope sites, and cloud integration environments, some of which are persistent, and others which are created on demand.

The PSIs are intended for testing specialist hardware, such as FPGAs, which we cannot provision within our cloud environments; General Processing Units (GPUs) are not considered specialist here, as they are available in our cloud environment. Testing in our cloud environments may necessitate configuration to use mocks or sims where otherwise a component would require specialist hardware; for all other software components, we expect to be able to do basic tests in the cloud. The tests in the PSIs and cloud environments include the unit tests for components, but also BDD tests for components and sub-systems, and can include combinations thereof.

After testing in either or both of a PSI and our cloud environments, individual components and sub-systems are made available to the ITFs (integration test facilities) in order to perform formal verification, first of individual products and sub-systems, and thereafter to verify that the integration of multiple sub-systems works correctly. There are two ITFs: one for each telescope, based in Australia and South Africa. There are three PSIs, in Australia, Canada, and the Netherlands, with different specialisations.

There are limitations to what can be achieved in the cloud environment and in the ITFs. The ITFs integrate multiple sub-systems relatively late, so the cloud environment can be used to detect issues before code is run in the ITF. The ITFs also do not have a powerful compute cluster, so they are unable to perform load tests of the control system, and cannot perform performance tests of the data processing. Indeed, full tests of the data processing cannot easily be performed on our standard cloud environment, and usually requires access to large HPC (High performance computing) clusters; however, there are tests that can be run in the cloud environment that cannot easily be run with the much more limited ITF resources.

The ITFs and PISIs can perform much better tests on the signal chain than the cloud environment can, as they can integrate FPGAs, and use signal generators and other specialist test equipment. Therefore, we are working towards a model where BDD tests can be used in the PSIs and cloud environment to test respectively the behaviour of components in the signal chain, and components in the control system, before verification (where that is possible) in the ITFs. The precise test steps needed in the ITFs may not be suitable

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC001

for use in the cloud environment, as they are primarily manual (*i.e.* directly triggered by a tester); however, we aim to converge on our terminology, to create our shared Domain specific language (DSL) that describes how to operate radio telescopes.

Without a DSL, there can be issues, as gherkin is not a precise language. Though the basic syntax is simple, steps can be defined with the full use of natural (English) language. Therefore, defining a scenario or step in gherkin has multiple possible ways to be interpreted in code, or even multiple ways to be understood in English. Steps can be defined using different terminology, but with the same intent. Therefore, discussion and convergence on and creation of a DSL are essential.

Despite the difficulties, the benefits that BDD can offer around specifying system behaviour, coupled with the long-term nature of the project (meaning that benefits can be realised over multiple years), encourage us to continue. Some of these difficulties, but also the benefits, are elucidated below.

## TUNING THE INTEGRATION PROCESS

On paper, the principles behind BDD testing, and the theory of using those inside the SKAO, look good, even though our testing environments are more complex than usual. However, we have encountered problems when implementing the process.

Our first prototypes of BDD tests happened over 3 years ago, at the level of an individual team, where one of us (VA) wrote the first gherkin definitions. Our inexperience with the testing technique and the tools to support it meant that we first tried it out at the unit test level, which we swiftly found did not work. These steps were implemented in Python, using the pytest-bdd module, and the results were reported through the XRray plugin to Jira, our work management system [16], thus prototyping a generic test management platform. For the initial experiment, at the unit test level, we quickly found that we were flooding Jira with test executions and tests that were not helpful to stakeholders wanting to understand the capabilities of the wider system.

The team subsequently refactored the tests to work at a component level or higher, and Jira now has a record of those test results in Fig. 2. These can be used by the PMs for the SDP to ensure there are no regressions, and are extended when new features are added to the software, without overwhelming the PMs with unnecessary detail. In subsequent PIs we have evolved our use of Xray and Jira for test management, based on this successful prototype.

### *Tango Testing Challenges*

When we tried to roll this out more widely over the next 8 PIs, we found a number of technical issues with Tango that made implementing BDD testing challenging. Firstly, every Tango device expects to be deployed with a TangoDB database with which it registers. This is annoying for unit testing, where it takes more time and more resources (either

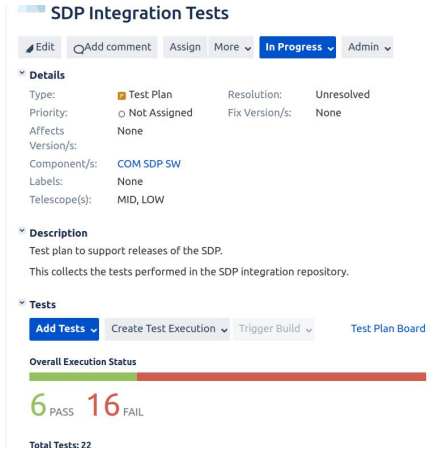


Figure 2: Screenshot of a report on the SDP integration tests, showing 6 passed and 16 failed tests.

on a developer's laptop or on a virtual machine in the cloud to spin up a database. The solution SKAO came up with was contributed back to the PyTango project [17]. Similarly, it is complex testing multiple Tango devices that interact asynchronously, as if the test requires a particular response, it may come after some other response from another device. This has been addressed [18].

We were also trying to develop a testware suite to help us with managing our state machine for some of our Tango devices. This ultimately failed as we under-resourced its development, as we discovered a gap in our resourcing mentioned in .

Finally, writing test fixtures for Tango devices is be challenging, as they are finite state automata. This means that for a test script to arrange to be in the state desired for the starting Given condition, one may have to pass through several other states in order to arrive there, unless one happens to be in a state in which it is valid to command or trigger that state transition. As an example shown in the model of one of our state machines in Fig. 3, in order to run a scan to collect data from the sky (by sending the SCAN command to one of our Tango subarray devices), one must be in the READY state. To reach the READY state, one must first have sent a CONFIGURE command, again from the correct starting state. Thus, if one is not already in IDLE, one must first perform the steps to reach that state before one can issue the SCAN command, unless one is testing specifically the system behaviour when it is sent the SCAN command and is not in a state where it is able to execute that command.

This issue is less often seen in other domains; for example, when testing an e-commerce site, one may want to set a state for a test where a customer is logged in, with a stored address for shipping. To do this, a customer object can be mocked. The mock can contain all the data needed, and it is a simple matter to write the test to use the mock, and make it look like there is a logged-in customer with a stored address. This can be used through multiple tests of customer actions, or a separate mock can be created to test alternate flows (when the customer wishes to update an address, for example). The

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC001

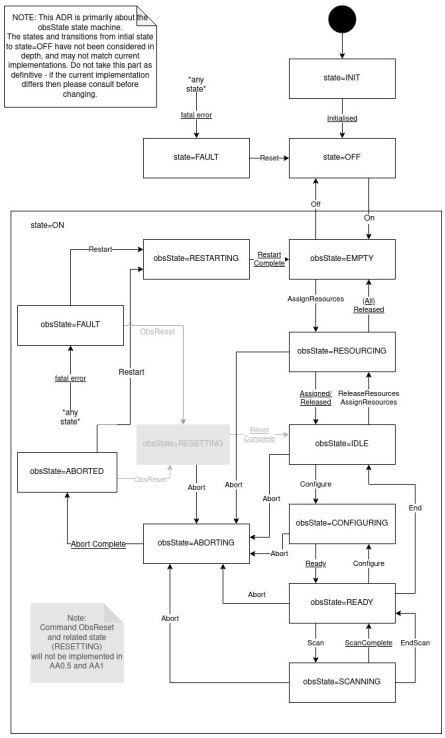


Figure 3: SKA Observing State Model

presence of other customer objects (whether mocks or real objects) will not affect the vast majority of tests.

In contrast, for a finite state automaton, we must force the components in the SUT through the states, including any data updates that are needed to arrive in a valid state, with valid data stored, for a given test. This also makes cleaning up after tests, whether passed or failed, more complex. This adds considerably to the complexity of the testware.

## *Working With Developers*

Meanwhile, in order to get developers started, we referred them to documentation on BDD testing, especially the Xray plugin for Jira. We provided a BDD testing guide and walk-through to provide information on our local context, in particular to enable them to connect BDD tests with Jira [19,20]. This was taken up by some teams, though with limited success, in part because of the issues described above, which we discovered during the roll-out process. We found that not all our PMs and architects were familiar enough with BDD to help write good test steps, though they could come up with good examples. We also found that where we did try to create BDD tests for new features in that way, teams then found that the tests were hard to implement, sometimes through poor formulation, sometimes because technical issues (such as those outlined above) made implementing test fixtures difficult. The teams also believed that they were not able to change these test step definitions; this was addressed by a discussion during a Testing Community of Practice meeting, where we explained that creating the gherkin step definitions was a dialogue, not something set in stone. While the end result has to be a specification of a behaviour, steps can be adjusted to facilitate test implementation. This appears to have improved the situation; the developers are not currently raising this as an issue.

## *Further Roll-Out*

Facing low adoption and a lack of understanding of what our system could do, we then tried to take a more top-down approach, to look at the behaviour of multiple sub-systems, as adoption at the sub-system and component level has been patchy. We started by formulating goals that we wished to achieve during PI17, and attempted to construct BDD tests based upon those goals. However, we found that the way the goals were formulated, and the way the software system was designed, made it difficult to define good scenarios. We also were defining the tests post-facto, based primarily on already-implemented code, so we could see none of the design-related benefits. Teams found it hard to own any of: the test step definitions; the test implementations; the testware development. We also found it hard to get good traceability between the tests that were being implemented and the features that were meant to achieve the goals. This meant that teams didn't see the point of the tests, nor did our major stakeholders see the links between the code, the tests, and the living documentation of system behaviour.

Therefore in the next PI (PI18, March-May 2023), we improved our goal definition, and began defining our test steps earlier, to improve the linkage between test definition and design. SKAO also started to use the reports in XRAY to better understand our tests. We still lacked good mapping between the tests and the features in all cases, and we still lacked team ownership of the integration process.

In PI19 (June–August 2023), we worked closely with the teams providing the TMC sub-system, where they improved the test harness they were using, to use some of the asynchronous testing techniques mentioned above, and to provide a factory for simulated tango devices. With that foundation, the teams could implement the infrastructure needed to run BDD tests, including defining the fixtures needed. Two of the authors and two TMC developers devised a set of scenarios for our subarray control device, which we then discussed with the OMC architects. This process revealed that we had not yet considered how the system should behave in some corner cases, which thus revealed more design work that needed to be done. The discussions revealed to the authors (who are testing specialists, but not experts in Tango, nor in the details of the SKAO control system design) some complexities about the state model; this will help inform UI (User Interface) design activities. The team implemented the scenarios where the design could be agreed upon, and uncovered a new bug during the process [21].

The BDD scenarios for the TMC are complex, as effectively these are scenarios for the behaviour of the whole control system. For the TMC to report on the health state of the telescope, it must aggregate the health statuses of the various sub-systems, which themselves must aggregate the health of their various components. Therefore, the scenarios look at what happens when various sub-systems are in

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BC001

the FAULT or DEGRADED states. We also consider what happens when various commands are issued, but not necessarily successfully completed by a given sub-system. This was the area where we encountered gaps in our design and implementation, which we can now follow up on using the SAFe processes.

The TMC teams are now equipped to extend their BDD tests as new scenarios are agreed upon, and to improve some of their existing BDD tests, based on the coaching given during PI19. They should also now be able to use the fixtures to test new features more easily, and also use the techniques they have learnt to help devise the BDD scenarios for those new features; we will be able to evaluate this at the end of this PI. The exercise, through the conversations between the teams, the architects, and the test experts, revealed gaps in our design, and helped fill in gaps in our understanding. The BDD tests now provide a record of the scenarios, showing the current capabilities of the system, and the associated Jira reports show the current test success (or failure), thus providing us with a set of regression tests.

While we have been successful with pairs of teams working together to perform interface testing, getting buy-in from multiple teams for system integration still eludes us. This may be easier in smaller and/or less distributed organisations. This is partly because unless teams happen to be in similar timezones, it is hard to arrange meetings and discussions as mentioned in . Because the system tests and testware are owned by everyone, there is also no direct authority for someone to say that they want a particular feature of the testware, or a particular reworking of a test. This may be easier in smaller and/or less distributed organisations, where it is easier to get a critical mass of people together.

## CURRENT STATUS AND FUTURE IMPROVEMENTS

We can see that the issues with implementing BDD testing, especially of the control system, fall into two categories, technical and social. While technical issues may be difficult, we have either solved them, have workarounds, or a clear path to improvement. The social issues around communication and adoption are harder to solve. Many of these issues are related to our complex system and our highly distributed organisation. However, the social benefits are where BDD can deliver its most profound benefits, by fostering discussion, finding examples, creating and using a common language, and providing living documentation. Hence we are motivated to persist, especially given the long lifetime of the project, which encourages us to make the up-front investment in quality, in order to ensure smooth operation and maintenance.

To address this, we are pursuing more point-to-point integration in PI20 (the current PI), where two teams integrate their components or sub-systems together, while we resource the work to co-ordinate multiple sub-system tests. This will extend the close work that was performed with the teams working on the TMC sub-system during PI19, and allow us to continue work on defining a DSL, so that we have a library of steps available for when we secure our resources. This will also allow some knowledge transfer between testing experts and the domain experts within the teams. While this is slow, slower than we would like, it is the best we can do with the available resources.

We are also organising hackathons, focusing on component and sub-system tests. We will run some remotely, with a format of two half days (to minimise the worst of the time zone related issues), first of all looking at test code in groups, then reviewing merge requests. We plan then to run a face-to-face hackathon at our in-person PI planning meeting in December 2023.

Without testing multiple sub-systems together in the cloud, we are accruing risk, as such complex systems often exhibit emergent behaviour, and it is vital to identify any misbehaviour before we deploy to a production system in 2024. Once resourced, we intend refactor our testware, first to re-enable tests of multiple sub-systems, and then to provide tools that can be used by the sub-systems as part of their integration testing. This would be trialled with the teams, to drive better adoption than we have achieved hitherto.

## CONCLUSIONS

BDD testing has a lot to offer in the context of testing control systems for complex instruments, such as the SKAO telescopes. We have already seen an improvement in the understanding of the tests, through the beginning of development of a DSL. We have detected new bugs, and uncovered a divergence between our design and implementation.

We have also seen that it requires considerable attention to the accompanying testware, and to the education of the people who will need to work with the BDD tests; this paper should provide some idea of the potential pitfalls other adopters may wish to avoid.

We hope that this paper inspires others to adopt BDD testing processes for their control systems, or at least provides them with the information they need to assess whether using BDD testing techniques is likely to be of benefit.

## REFERENCES

[1] SKA Mid Telescope, https://www.skao.int/en/explore/telescopes/ska-mid

[2] SKA Low Telescope, https://www.skao.int/en/explore/telescopes/ska-low

[3] N. Rees, "SKA Project Status Update", presented at the ICALEPCS'23, Cape Town, South Africa, Oct. 2023, paper FR1BCO03, this conference.

[4] Tango, https://www.tango-controls.org/

[5] Scaled Agile Framework, https://scaledagileframework.com/

[6] G. Brajnik, M. Bartolini, N. Rees, "A sustainable software testing process for the Square Kilometre Array Project", in *INCOSE Int. Symp.*, 2023, vol. 30, no. 1, pp. 109-123.

[7] Reduce the average time for delivery of software changes, https://jira.skatelescope.org/browse/SPO-2673

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-MO4BCO01

[8] Communities of Practice, https://scaledagileframework.com/communities-of-practice/

[9] Software Testing Policy and Strategy, https://developer.skao.int/en/latest/policies/ska-testing-policy-and-strategy.htm

[10] Certified Tester Foundation Level Syllabus, pp. 27-8, 2023. https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB_CTFL_Syllabus-v4.0.pdf

[11] PACT, https://docs.pact.io/

[12] BDD Testing, https://www.bddtesting.com/

[13] J. F. Smart, J. Molak, "Chapter 1", in *BDD in Action*, Shelter Island, New York: Manning, 2023.

[14] G. Adzic, "Chapter:1 Key Benefits", in *Specification by example: how successful teams deliver the right software*, Shelter Island, New York: Manning, 2011, ch.1.

[15] SDP scenarios, https://gitlab.com/ska-telescope/sdp/ska-sdp-integration/-/blob/master/tests/features/subarray.feature?ref_type=heads

[16] Xray - native Test Management for Jira, https://www.getxray.app/

[17] Device Proxy, https://pytango.readthedocs.io/en/stable/client_api/device_proxy.html

[18] D. Devereux, "Asynchronous Testing of Tango Devices in SKA", presented at the ICALEPCS'23, Cape Town, South Africa, Oct. 2023, paper MO4BCO03, this conference.

[19] BDD Testing guide, https://developer.skao.int/en/latest/tools/bdd-test-context.html

[20] BDD Walkthrough, https://developer.skao.int/en/latest/tools/bdd-walkthrough.html

[21] TMC scenarios, https://gitlab.com/ska-telescope/ska-tmc/ska-tmc-integration/-/tree/main/tests/integration/tmc_harness?ref_type=heads