Situation coverage – a coverage criterion for testing autonomous robots

Rob Alexander*, Heather Hawkins*, Drew Rae [†]

[*] *University of York, York, United Kingdom*

[†] *Griffith University, Brisbane, Australia*

rob.alexander@york.ac.uk

Keywords: situation coverage, autonomy, testing, environment, context.

Abstract

Autonomous robots (AR) can get themselves into a wide range of situations, and they do not have a human to look after them in fine detail all the time. When we test autonomous robots, we must therefore care deeply about the range and diversity of the situations in which we have simulated and tested them – we must make sure that the *situation coverage* of our testing is adequate. Situation coverage measures can be implemented quantitatively, and so unlock a range of automated testing strategies. There are epistemic challenges to justifying the confidence we should attach to test results driven by situation coverage, but they are not fundamentally more difficult than those faced by other coverage criteria.

Technical Report, Department of Computer Science, University of York

Report number YCS-2015-496

Revision History

1.0	11 Feb 2015	First public release
1.1	7 April 2015	Minor wording and formatting improvement

1 It is hard to derive adequate safety requirements for autonomous robots

> The courier robot rolls through the suburban streets, its heavy cargo of Amazon books, greetings cards and groceries gives it a substantial mass. It dutifully stays under the speed limit, gives way to other drivers, all the while counting down the house numbers as it heads for number 51. Suddenly, around the corner from Acacia Avenue comes an ice-cream van, grotesque in its lurid colours and its horrendous cacophony sounding. On top of the van is mounted a menacing object, roughly consistent with either a surface-to-air missile or a tank gun. The courier slams itself into panic evasive mode, and screeches across the opposing lane, onto the grass verge and through a garden fence. It is badly damaged, but at least no-one is hurt.

Or…

> Another courier rolls along a fast road cutting through the city, scanning carefully for the icy surfaces that could result from the current cold temperatures. Ahead, a careless driver crashes into another car and a dangerous pile-up spins up, but it's ok because the courier can see a vector where it can go off road and skid to a halt just before it reaches a large crowd of children in heavy coats. The courier veers off the road, adjusting its driving dynamics appropriately just as it first touches the grass – the soft, white, powdery grass which mysteriously doesn't behave like grass should at all. The courier wheels don't behave as they should, and even with the brakes locked solid it just keeps moving. As it skids towards the children it flashes its warning lights. All they can do is watch, rabbits in headlights, as the robot tumbles towards them. One of them drops the sledge.

Autonomous robots (AR) can get themselves into a wide range of situations, and they do not have a human to look after them in fine detail all the time. They must therefore react to those circumstances in a way that is sensible, or at least not disastrous. A courier robot that reacts to an ambulance by moving to ram it is a poor courier and a public hazard.

As Dogramadzi et al. suggest in [1], the majority of challenging situations that AR will find themselves in are less part of their mission than they are part of "getting by" in the world – dealing with the environment and staying out of trouble. It is easy to forget about these situations, or never become aware of them in the first place, if you don't use analysis and testing specifically aimed at finding them.

If the above example vehicles had been tested using a *system coverage* approach – if they had been tested until some set of system components (software functions, modules, paths, branches) had all been tested at least once – these issues would not necessarily have been identified. They refer to external situations – situations that may not be addressed at all by any system component.

If the above example vehicles had been tested using a *requirements coverage* approach – if they had been tested until their conformance with all identified requirements had been tested adequately (for some reasonable definition of adequately) – these issues would not necessarily have been identified. The snow-handling example refers to a terrain type that in this (hopefully unrealistic) example might not been identified as a requirement for handling; it represents a failure of validation (with respect to reality) rather than verification (of system against specification). Requirements-based testing is primarily a verification activity, so might well not have found these faults.

If the above example vehicles had been tested with some form of s*cenario coverage* – where the aim was to cover a representative set of scenarios – these issues would not necessarily have been

identified. It is likely that a relatively small set of scenarios would have been explored, and that these would have been explored in a narrow way. Adequate testing of AR crucially requires that we study not just linear scenarios but the dynamics that arise from the interaction of the AR with its environment.

When we test AR, we must therefore care deeply about the range and diversity of the situations in which we have simulated and tested them. We need to test them on roads, in the rain, and with people in the way. We need to test them when they're in intermittent supervisor contact and when they've got a sticky wheel. We need to test them in *combinations* of all of those cases. We should, as part of our decision to declare said robot viable and safe, become confident that we have explored an adequate range of situations. We can refer to a measure of this adequacy as *situation coverage*.

Situation coverage is particularly important because it is hard to derive adequate safety requirements for AR. As ever, we know what harm looks like (e.g. a robot car runs over a pedestrian) but we do not know all the paths that can lead to that harm (e.g. it might be that it cannot distinguish between humans and lifelike statues, so after going into a skid it chooses to hit one pedestrian rather than knock over three mannequins). We are unlikely to generalise well from our experience with existing systems; AR are not like existing systems in that they are neither humans nor traditional software, and we are working on (and expect) lots of novel, adventurous AR uses (e.g. swarming sensors, ultra-long-duration high-altitude monitors, crawlers finding earthquake survivors). This fusion of novel technology and novel use is a classic analysis-challenging circumstance (Def Stan 00-56 users are referred here to the McDermid Square [2] – we are in the most difficult quadrant).

It is likely that conventional analysis techniques will help us only so much. AR software is necessarily very complex, and as noted above it must respond well to a very wide range of environmental stimuli (indeed, it must cope not just with stimuli but with situations – it must continually respond to stimuli so as to control itself and its environment and to maintain safety constraints). Techniques such as the Bristol Robotics Laboratory's *Environmental Survey Hazard Analysis* [1] have potential, but as purely manual techniques it is not clear to us that they will, alone, be able to bypass the assumptions of the testers about what will be challenging and what will not. Linear-scenario-based approaches may not be particularly insightful, as much of the interesting behaviour here happens once the AR is off the nominal scenario course and having to "creatively" respond to challenging circumstances.

It is likely that we will want to supplement the above techniques with automated analysis and testing tools, and many forms of such automation need to be guided by some explicit criteria.

2 Situation coverage methods can help guide us to adequate requirements

In conventional software testing, there is already a great tradition of coverage measures – the most basic being "did each line of the software's source code get executed by at least one test?" Aside from the criticisms of many of them as software coverage measures (and there are many such criticisms), such pure system coverage measures may well miss external circumstances that matter. For example, if we follow Thompson [3] and set out to test an AR in all the circumstances its software treats differently (e.g. if the software distinguishes dry weather, wet weather, and icy weather, and we test under all of those) we may not realise that it does not handle some circumstances at all (e.g. that the software has no sensible plan for dealing with snow).

We can adapt the ideas of software coverage to situation coverage – we can aim to cover enough situations, and a diverse enough set of them. We can attempt to quantify this.

If we have a concept of situation coverage, and metric or metrics that measure it, then we are better placed to "validate" our requirements – to discover the set of mid-level behavioural requirements that we need. Just enumerating the components of situations that can be encountered (terrain, weather, peer actors, "missions"…) may cause us to consider more cases (cf. the typical experience when formalising requirements – the rigour there forces clarity).

A common concern when defining tests is that the requirements errors made by the testers (e.g. invalid assumptions, misunderstandings of intent, superficial models of external phenomena) may be the same set of requirements errors made by the programmers – the tests may thus have blind spots exactly where the programs have faults (cf. the Knight-Leveson experiment [4]). Defining a situation space, and then covering it by generating situations, is a very indirect way of defining tests, so there is more chance that analysts will *not* make the same wrong assumptions that others made when designing the system.

When we specifically consider *simulated* testing, as much AR testing of course will be, defining metrics for situation coverage can help us unlock the full power of modern automated testing techniques. Although CPUs aren't getting much faster, they are still becoming cheaper, so automated testing is becoming more and more powerful. Given a good metric, automated testing can work very well. Without a metric, such techniques are often quite limited – no metric means no heuristic search or similar techniques, so we are reduced to random (or brute-force exhaustive) test generation.

The level of rigour we suggest here is perhaps best related to that of DO-178C and similar – we are proposing rigorous testing using a systematic coverage measure (analogous to 178's use of MCDC coverage for software), not the kind of absolute completeness that formal methods often strive for.

Of course, we are not seriously suggesting that anyone should use *only* situation coverage to guide their testing. Relying on a single form of coverage criteria is unwise, and relying wholly on coverage criteria for test generation is also unwise (on the latter, see Gay et al. [5]).

3 Situation coverage measures are practical

There are a variety of ways that situation coverage could be implemented – entities and situation elements can be enumerated and combined in a variety of ways. Experimental design techniques can be used to choose critical combinations, and sensitivity or principal component analysis techniques can be used to prioritise them.

Broadly, situation coverage measures can be approached from a Macro- or Micro- perspective. Macro-SC means looking at the situation overall and saying whether certain things are covered. For example, we can look at the map and say that there are trees on it, therefore trees are covered as an entity type. Micro-SC means that we watch the system run and see what small-scale situations are encountered. For example, we can "watch" the AR and note that it does indeed enter the situation "following another vehicle at a fixed distance". We can of course use macro as an approximation to micro (we look at the map and mission, and note that there is a stationary car blocking the lane that the AR would naturally take to its objective, so a micro-interaction of "overtakes a parked car" is very likely to occur).

At the macro scale, we can build situation coverage metrics based on entity types (e.g. is there a car? A tree? A humpback bridge?), number or density of entities of given types (e.g. "there is a steady population of 15 other cars in the village"), relationships of entities (e.g. "on the same radio net"), mission types, and many more. The list is almost endless; the trick is to identify which metric elements are of high value for testing.

At the micro scale, we can cover types of conflict (e.g. head-on, same course but slower, in wrong lane…), small-scale relationships (e.g. following, overtaking, observing across a central reservation), paths taken (including analogues of path and branch coverage from software testing, where the paths and branches are literally roads and junctions on the map) and many more. Again, the challenge here is prioritisation.

Given the metric components above, we can combine them in a variety of ways – hit every coverage element (e.g. perform at least one run containing each entity type), hit every N-way combination of elements (e.g. perform at least one run that has trees and wall, one that has trees and fences…), and so on; the software testing can furnish us with a wide variety of methods for combining elements. Probability can be included (e.g. attempt all situations down to estimated probability 1×10^{-6}) and multi-objective optimisation techniques can try to maximise multiple values (e.g. get the AR to have as many low-speed collisions as possible with the most cautious other drivers).

As with all testing, our choice of what to cover and not to cover is implicitly based on a *uniformity hypothesis* – when we say that two situations are equivalent for coverage purposes, we're saying that the AR will behave the same under both, or at least will have the same failure behaviour.

Analysis based on situation coverage can and must, of course complement existing approaches. Automated testing is particularly important because, as noted earlier, AR are going to be extremely complex systems and will need an enormous amount of testing. Situation coverage can help guide that testing.

Situation coverage can, in theory, support existing requirements analysis approaches. It's potentially a validation approach, which makes it complementary – it can very plausibly show you that your current AR behaviour specification doesn't give you the resulting behaviour (in context) that you desire.

Implementing testing based on Macro-SC is very easy to do – simple input space partitioning (see Grindal et al. [6] for a survey) and experimental design techniques can be applied. Micro-SC is harder to meet (because it involves predicting what will happen during test runs, based only on their initial

conditions) but there are many applicable techniques for this (e.g. see the recent work by Poulding et al. [7]).

For “raw materials” (situation elements) we can build on ontologies, including interaction ontologies [8]. All of these can be improved over time to meet our new situation coverage needs.

4 There are epistemic challenges to situation coverage, but they are not fatal

Software coverage has been variously criticised, and with good reason – it is yet another example of a metric to obsess over, to become obsessed with, and to fetishize, at the expense of bringing our experience and intuition and case-specific reasoning to the testing problem at hand. See Kaner et al. [9] for extensive discussion of this.

Combined with automated test generation, coverage criteria are particularly pernicious, because they are definitely not enough *on their own*. It's one thing to use them on hand-crafted tests, the original design of which was not primarily motivated by the criteria, to sound out something you might have missed (by not covering it at all). It's quite another thing to let a computer generate tests *solely* guided by the criteria. There have been a number of demonstrations of this problem – see Gay et al. [5] for a summary.

The Macro-SC discussed earlier is particularly vulnerable to this, and Micro-SC provides a potential antidote – it ensures that certain patterns of interaction do in fact occur. For example, it may be possible to "trick" a situation coverage criterion by presenting a patchwork of tests that each cover a single coverage element but are otherwise very unchallenging for the AR. This is a particularly acute problem for situation coverage as there is no "natural bound" on the test; in contrast, when tests are defined by their covering a requirement, the criterion for a "sufficiently thorough individual test" is that the requirement is triggered and that adequate time (etc.) is given to observe the results. Similarly, when the unit of testing is the scenario, the unit of individual test adequacy is that the whole test is completed. It is not obvious how to define such adequacy criteria for situation coverage, other than by using additional criteria to drive the adequacy of each test, and then applying situation coverage as an additional check of the overall test set.

Coverage weaknesses are a particularly acute problem for safety-critical software, because you're inevitably going to make some pretty serious claims based on the results. You can see these weaknesses in the criticism that's been applied to MCDC testing – the explicit rationale for using it, for trusting it in a safety-critical role, is extremely flimsy.

Yet DO-178C still mandates MCDC. It is justifiable, in context, because it is just one part of a broader safety effort. We do not rely on MCDC alone; this paper does not propose that we rely on any situation coverage metric, no matter how sophisticated, alone.

So, as ever, it will remain necessary to target multiple test coverage criteria simultaneously. When automatically generating tests, situation coverage can be used as the primary criterion, or can be mixed in with others. Similarly, when tests are hand-crafted using informal and largely implicit criteria, situation coverage can be measured as a check of what coverage has been achieved. Situation coverage is one approach to assessing the adequacy of AR testing – it is not sufficient on its own, but it is valuable as part of broader assessment.

We can build strong safety claims from many complementary weaker claims. The classic example is for software – there are no grounds from testing alone to support software dangerous failure rates below 1×10^{-4} per hour but it can contribute to claims several orders of magnitude higher [10].

In any case, it is impossible to avoid the need to claim situation coverage. When you assert that your tests (of software alone, or of the whole system) are adequate, you are asserting that the system has been tested in an adequate range of situations. By systematising this, by having actual situation coverage metrics, we can refine this process and make more valid claims. We can study the whole issue better.

A situation coverage metric without a theory behind it is of little value – a coverage criterion cannot be convincing unless there is reason to believe that the way it subdivides the world is valuable, i.e. that the uniformity hypothesis (see earlier) is at least close to true.

There are a number of grounds on which a supporting theory could be built:

- It could be based on past experience, e.g. on what entities and combinations have caused trouble for similar systems in the past.
- It could be based on an abstract model of an AR, such as the NASA Goddard Agent Architecture as used in [1]. This would then support a uniformity hypothesis based on the uniformity of significance of events/situations for this abstract model.
- It could be based on a systematic refinement of an environment from the very abstract to the very concrete, perhaps in the vein of Ontological Hazard Analysis [11].

We can also evaluate situation coverage measures empirically, although we are unlikely to be able to support safety-critical claims through this alone. We can seed faults into the AR (perhaps through mutation testing techniques [12]) and evaluate how well SC-guided testing finds the behaviour caused by the faults.

All test coverage measures, and all requirements analysis measures, raise difficult issues of epistemology. Being at the intersection of those two areas means that situation coverage has particularly difficult epistemic problems. We believe, however, that testing guided by situation coverage will be a cost-effective way to find faults in AR, particularly vehicle-level specification faults, and that empirical studies will bear this out. We also believe that, over time, AR developers will be able to build plausible theories to support their specific SC criteria.

5 Useful situation coverage must achieve certain things

A situation coverage criterion that is to be used as part of a safety argument needs to be *based on a theory that justifies its adequacy*. A theory allows generalisation from the cases we have tested to the cases that we haven't – it justifies the uniformity hypothesis for that test set (it justifies treating certain subsets of the tests we didn't do as equivalent to the tests we did do).

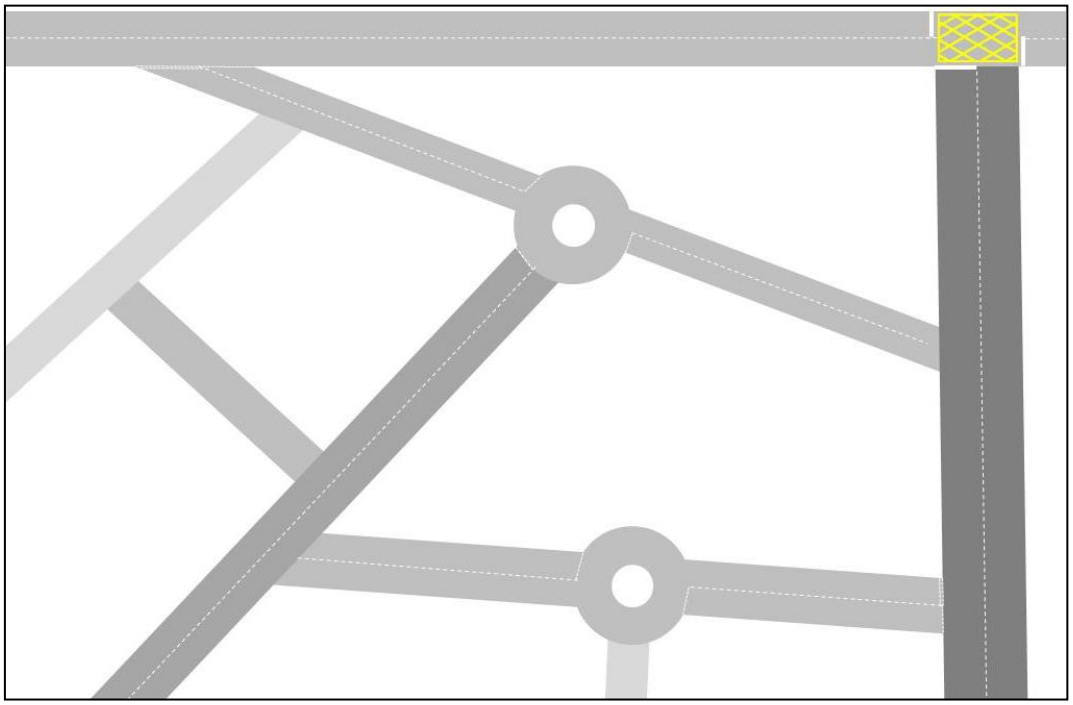
A situation coverage criterion needs to be *empirically* evaluable, and this evaluation needs to be performed. It is not practical to empirically demonstrate the adequacy of a criterion at safety-critical levels of integrity, but is certainly possible to empirically show its limitations (for an example, see Baker and Habli's [13] use of mutation testing to show weaknesses in DO 178B-compliant test sets; Hadley and Clark [14] similarly use mutation testing to challenge MCDC specifically).

For use in many forms of automated testing, a situation coverage criterion needs to be *quantifiable, and able to be measured*, at least from a simulation where ground truth is available. The challenge here is in moving from an abstract idea of "we need to check that we've covered this set of situation elements" to actually having an algorithm that can implement the criterion.

6 Situation coverage metrics and the use thereof – a simple example

Basic Example - One Criterion

Consider a simulated AR in the form of an autonomous car in a road system. Imagine that simulation is reasonably sophisticated e.g. roads may be at arbitrary angles to each other, they may vary in width, surface type (and condition), camber, precise road markings (e.g. the "do not overtake" centre line). It supports roads joined by box junctions, roundabouts, mini-roundabouts, and unmarked, unsignalised areas that just happen to have several roads attached. At junctions it can distinguish between major and minor roads (and provide signs and road markings to indicate that) or leave it ambiguous.

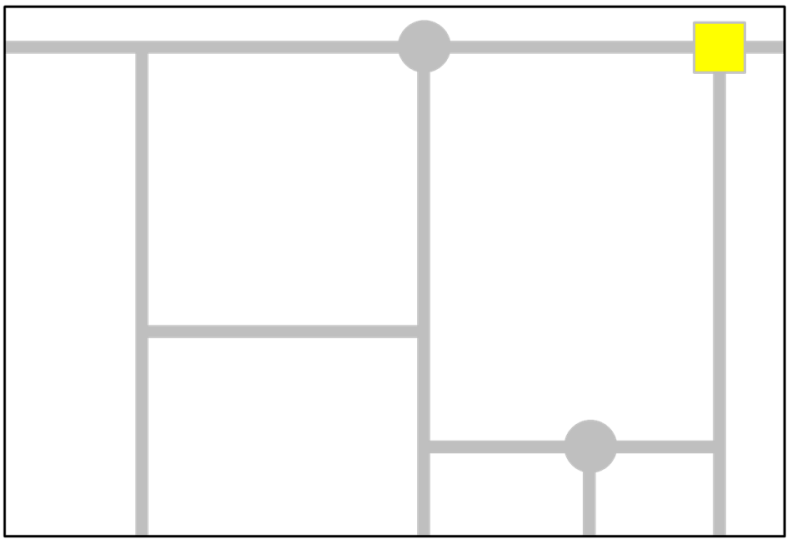


We could make a coverage metric for "all simulatable road systems", but even with some arbitrary restrictions e.g. the area covered by the system, and the total length of the roads, the space of this is enormous - practically infinite for our purposes (certainly if e.g. floating point angles and lengths are used).

There are various ways we can abstract from this model. Here, we will do so by representing the road system as a graph, a set of junctions (nodes) and roads between them (edges).

The graph can be topographic rather than simply topological, and we can require that the edges be oriented only at a restricted set of angles – for simplicity, in this example they can only be angled in the four cardinal directions. When representing a simulation map as a graph, roads would be assigned to whatever cardinal direction is closest - if this leads to several roads being at the same direction, they would be treated as a single road[1]. (In real use, we'd want at least 8 directions, possibly 16 so as to capture some of the sharper hairpins that occasionally happen e.g. on country lanes).

[1] This is probably not as simple as it sounds – we could lose a fair amount of the complexity of the network this way, and possibly change traffic patterns quite a lot as some routes could be deleted by this method (and those routes might happen to be heavily used).



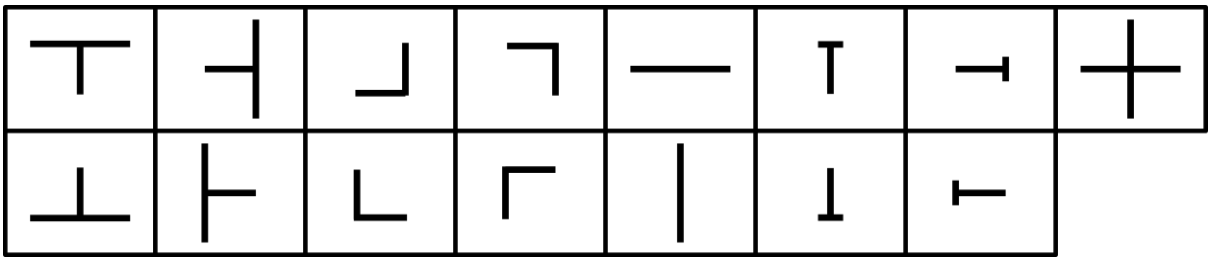
When making or choosing a test coverage criterion, we want one that is *tractable.* There are two ways to look at this:

- Given a system under test (SUT) and a set of tests, can a coverage percentage be calculated?
    - Examples that *don't* satisfy this include:
        - All-paths coverage of a program with loops (there are infinite paths)
        - Percentage of SUT faults triggered (because the full set of faults cannot be known with certainty prior to exhaustive testing)
- Given a SUT and a set of tests, can 100% coverage be achieved under realistic practical use conditions?
    - Examples that satisfy the previous condition, but not this one, include:
        - Many dataflow criteria, when applied across procedure (etc.) boundaries in non-trivial programs

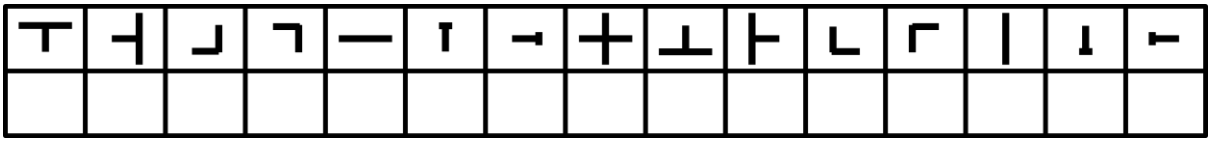
In testing, it is common to use the term "intractable" when the former is true but the latter is false i.e. when a percentage can be calculated but 100% is not realistically attainable.

Even if we limit the number of nodes and or edges in any given map, that gives us an enormous number of possible maps. An "all maps" coverage criterion is thus intractable (in the second sense given above), even with the graph simplification. (An "all paths through the map" criterion is even worse). So any coverage measure that's tractable will abstract away again from this.

For this example, we can make a simple criterion that gives us a tractable space (indeed, one we can actually draw). The criterion is that the map set includes all possible junction shapes, of which there are 15:



Given some test situation we can say which shapes are covered; given a set of test situations we can measure the proportion of the shapes that are covered, with the aim to build a set that covers all of them. With this in mind, we can represent the criterion as a set of boxes that need to be ticked in order for a given map set to meet it:



A Further Criterion

Imagine, now, that the simulation includes pedestrian and other vehicles, of a wide range of types and behaviour profiles. For example, for pedestrians it has generic adults, drunk people, children, elderly people with several different disabilities, “crocodiles” i.e. groups of many people moving in a rough line. It supports a similar range of driver profiles. In terms of vehicles, it has vehicle and driver models for cars (commuter, boy/girl racer, drunk, learner, elderly with a driving licence but would not pass a driving test now...), HGVs (articulated or single-piece, over-wide vehicles with small houses on the back), bicycles (commuter, child, racer), ambulances, parade floats, horses, sheep…

Much as with the road layouts, we can’t use “all peer sets” as a criterion - we probably can’t even do “all possible individual peers” (imagine that all vehicle and driver models have numerous tuneable parameters, and possibly arbitrary scripting features). So we need to abstract our coverage criteria away from that.

We can require that the SUT interact with each “type” of entity, where type is defined as a narrow subset of the possibilities. Here we will say – car, bicycle, HGV, pedestrian all using a standard competent adult driver model (“interact with”, here, would be operationalised as something like “be placed in a situation such that the SUT must take action to avoid breaking some traffic rule with respect to that peer e.g. must change course to avoid collision or excessive proximity”. Actually implementing the in-simulation detector for that could be difficult.)

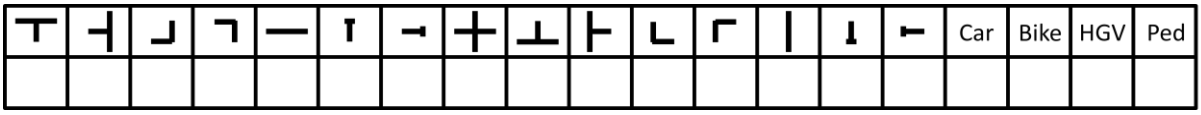
If we wanted a further peer-related criterion, we could take a cross-section of behaviour profiles - perhaps expert driver, sub-test-standard (e.g. very elderly) driver, drunk – and say that the SUT must interact with each, also.

We can represent this as a set of boxes, as before:

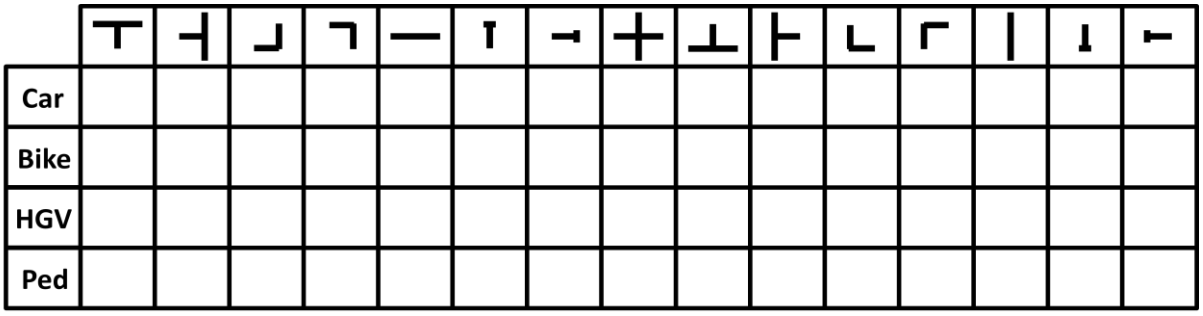
Car	Bike	HGV	Ped

A Composite Criterion

Given the two criteria above, we can combine them. The simplest way is to say “the SUT must interact with each entity type and each junction shape”, which gives us (15+4=19) boxes, thus:



This method is concise, as it doesn’t scale very fast with increasing junction model complexity or with a larger set of entity types, but it’s probably not that useful. Instead, we can define a criterion “the SUT must interact with each entity type **at** each junction shape”. (This gives us 15*4=60 boxes):



Note: this composite thing is "a criterion" – it is a rule by which a test set (here, a situation set) can be judged as complete or not. Like most useful criteria, it also allows a proportion of completeness[2] to be assessed - the proportion of boxes hit is the proportional completeness. The two things that compose it (which were individually described in the previous sections) are likewise each a criterion.

In Theory, What Might This Lead To?

If we are lucky, there might be some bugs lurking in some of the boxes.

- For example, it might be that our SUT does not, in fact, allow for the length of vehicles, nor their turning circle. Perhaps in the case (HGV ⌞) with the HGV approaching from the top/north and the SUT from the right/east (note how the criterion as specified currently does *not* say anything about these path-of-entity factors), the SUT may attempt to combine its turn with that of the HGV (i.e. turn right exactly as the HGV turns left), not allowing for the fact that the HGV will take longer to clear the junction and is likely swing its back wheels out further. It may risk a collision (or near miss that requires a sudden evasive manoeuvre that is itself risky).

- For example, it might be that our SUT basically assumes all vehicles follow the same speed and acceleration patterns. In the case of (Bike ⊢), with the bike going from south to east and the SUT from east to north, it may be that the SUT will again try to combine its turn optimally such that it crosses the box just as the bike is clearing it. As the bike moves slower than the SUT expects, the SUT may commit to a manoeuvre that risks a collision or requires risky evasion.

We can't say that these failure cases wouldn't be found by other means. Nor that the junction interaction criterion would definitely find them - they require other conditions identified above (i.e. directions of entry and exit) that are not part of the criterion, and even then they may only manifest if the timings, junction sizes and approach speeds, etc., have suitable values. You could tick all sixty boxes without running suitable cases. But the point is that the criterion requires that we run test cases which have *some* of the properties required to reveal these bugs.

In Practice, is this Criterion Actually Likely to be Any Use?

This is a more difficult question than the previous one, and there are two levels of "yes":

1. It leads us to find some bugs that might have been found by other means, for about the same cost
2. It leads us to find some bugs that probably wouldn't have been found

Failing to meet even the first level means it's actively worse than its rivals, one of which is "no particular test adequacy criterion at all".

Of course, a coverage criterion by itself cannot reveal bugs - it can only do so when paired with a test generation strategy which in some way is guided by that criterion. Furthermore, naive application of any tractable criterion can easily devolve into ineffective testing that meets the criterion in the absolute minimum way - they cover the precise criteria but don't actually achieve much worthwhile testing (i.e. *practical* coverage criteria are only ever a check on the adequacy of our testing – we can never, in practice, spell out *everything* that our tests need to do – see Gay et al. [5] and Hadley and Clark [14]).

[2] As usual for coverage criteria, the confidence of adequate testing we should take from percentages less than 100 is quite unclear.

The precise nature of the test generation strategy, therefore, is also of critical importance. Situation coverage is but one component of good AR testing.

Ahead-of-Run versus In-Run Coverage

In the above, we've glossed over one issue - do we assess the coverage achieved by a set of situations before we run them (Ahead-of-Run) or after we've run them, taking account of things we measure during the runs (In-Run).

In the example here, Ahead-of-Run (AoR) is quite limited, in that about the best we can do is change the composition criterion to "all junction shapes have been on the map with all entity types" i.e. having a ⊢ on the map and an HGV on the map ticks the (HGV ⊢) box. In contrast, In-Run (IR) allows us to score whether the SUT really entered a junction of a particular type, and whether it interacted with an entity of a particular type in that junction. Until that happens, no box is ticked.

Generally, therefore, an IR approach is more powerful.

7 Some likely questions and some reasonable answers

Why would requirements-based testing not give adequate coverage?

Situation generation based on situation coverage is a complement to requirements-based testing.

Kaner et al. [9] say (item 48 in their book) –

"If someone tells you to do 'requirements-based testing' she might be talking about any combination of these three ideas

- *Coverage (Test everything listed in this requirements document)*
- *Potential problems (Test for any way that each requirement might not be met)*
- *Evaluation (Design your tests in a way that allows you to use the requirements specification to determine whether the program passed or failed the test.)"*

Taking our definition as "all of those, simultaneously", we also need to consider what *level* of requirements we cover. It's likely that we will have some very high-level safety requirements ("Do not contact any human while moving at a speed greater than…") and some lower-level derived requirements that decompose them into something implementable ("Within 0.1s of detecting a solid not-permissible-to-damage object on a 2-seconds-to-collision-trajectory a 'maximum braking' command must be sent to each wheel motor").

We can safely say:

- Coverage – situation coverage covers external situations based on our knowledge of those. We do not propose basing our choice of situations on any requirements specification. So our coverage goal is different.
- Potential problems – we *are* looking for ways that requirements might not be met, but with respect to high-level safety requirements rather than to detailed requirements for the AR's behaviour. So we are looking for slightly different problems.
- Evaluation – again, we use the high-level safety requirements to provide our pass/fail criteria. So we have slightly different evaluation criteria.

For the *lower-level derived requirements*, SC-based testing is a means of evaluation/validation for those requirements - it has a good chance of showing that the requirements are wrong. That's something that requirements-based testing is not well placed to do. *We suspect this will be the main benefit of SGen-SC over requirements-based testing.*

Why would testing of scenarios identified by the safety analysis process not give adequate coverage?

Testing based on safety analysis is indeed a strong rival – it can lead engineers to recognise that they have missed a requirement that will be needed for safety.

Our approach may, of course, beat weak safety analysis approaches. And from our limited experience it is clear that *some* industrial practice is weak (also, sometimes, the explicit methods engineers use don't actually help them very much – the fact that they do well is down to their individual expertise).

Most explicit safety analysis – at least the kind that is likely to reveal challenging unforeseen aspects of the environment – is a paper-based, manual process. It doesn't scale particularly well to complex combinations of events and situations. An automated, simulation-based process, guided by situation coverage, may do better.

It's actually very hard to answer the above question directly and convincingly, since it's very costly and difficult to do any empirical or data-gathering work on *real* projects (e.g. we are unlikely to be

able to get access to AR-developer data regarding which kinds of faults slipped through what kinds of testing).

For In-Run / Micro-SC (where the SC refers to events that happen during a run, e.g. "the SUT follows another vehicle through a series of turns"), why not just generate those situations up front i.e. fill in the "boxes" of the SC one-by-one by generating situations to order?

This would be a good idea, if you can do it. But doing so may be difficult - you may not be able to code an algorithm in reasonable effort that generates those situations deterministically and efficiently. As is often the case, it's easier to declare what you want (here, cover some specific SC criteria) than to write a good algorithm to generate it. This is likely to be particularly true if your SC asks for more complex interactions or event sequences.

The standard computer science fallback in such a case is to use a metaheuristic, such as evolutionary search, to throw paint at the wall until our goals are met.

If we *can* write a deterministic algorithm that will generate the situations in each box, then we should probably do that. However, as mentioned previously, we know that even for well-established software structural metrics, really aggressive test generation based on metrics alone gives dubious tests – the tests generated might cover the precise criteria specified, without actually providing sensible testing.

So, we want to test more than those precise situations that the metric calls for - we want to at least cover those while covering some other SUT-in-world behaviour as well. At the very least, we want some random variation in there (e.g. we might want each "box" in the SC to be 'hit' ten times by runs that are different to each other (e.g. different random generation)).

SC can also be used as an additional check on situation sets generated by other means (much as you can use path/branch/MCDC/mutation to check the adequacy of a hand-crafted test set). This is perhaps where it will be most valuable.

If we use an evolutionary search technique that targets SC, is this really likely to be better than (or at least complementary to) a search-for-difficult-situations model where you define some difficulty/complexity metrics (e.g. *"number of junctions between start and destination"*) and target those instead?

It may be.

One reason it *might* - we don't always know *what's* difficult, so we can't be confident of identifying a good enough set of difficulty metrics. After all, if we could do this perfectly then it would imply we know *nearly* where all the bugs are. That would be an unusual state of affairs, to say the least. Diversity of testing is important because of this lack of understanding.

Of course, infinite diversity requires infinite effort. We need to justify our SC somehow, too.

NB doing a difficulty metric *doesn't* necessarily require specific knowledge of the SUT - you can in theory do this on a more abstract model of a cognitive agent or similar. But to do it really well would probably require SUT-specific knowledge (see the following comment and response).

Shouldn't our choice of situations depend on not just what situations are possible to encounter but what situations are likely to cause problems for our specific SUT, given its precise nature e.g. what we know about how it's coded, the maturity of certain components and the technologies they depend on, etc.?

In practice, yes. This is a factor we could consider when choosing SC metrics in real development, SC is one type of criterion for a "good situation set" - others include expected difficulty, requirements coverage, system component coverage etc.

For our immediate research, we are studying SC on its own. Eventually, we or others should investigate whether adding SC to an already fairly mature test process (which already has requirements coverage, code coverage etc.) is likely to increase testing power.

It is possible that SC may work when you don't understand the SUT that deeply. One potential manifestation of this is that it could find bugs that you don't suspect are there at all, including bugs that don't violate your derived requirements/vehicle behaviour specification but blatantly violate your high-level safety requirements.

Why is metaheuristic-search-for-SC (etc) likely to give superior SC to random approaches?

With any random generation approach, there will be biases towards some situations and away from others. Whatever algorithm you use, certain situations will only be reached by unlikely combinations of random results - more so for more complex algorithms (and no algorithm that generates plausible situations is likely to be simple). Good SC is likely to require algorithms that explicitly search for it.

But aren't those situations that *do* get missed by a random generator likely to be the highly improbable ones anyway (especially if you shape your random generation using operationally expected probabilities)?

Yes. However:

- Even if we set out to guide generation by operationally expected probabilities, in a plausible way, we'll be at least slightly wrong – some of our very rare situations will be more common than our generator suggests. If nothing else, we'll need a substantial probabilistic "buffer" before we declare something "too improbable to worry about given our safety targets" (e.g. a 10^{-6} target might require a 10^{-8} estimate before we're confident).

- As with ordinary software testing, we're unlikely to be able to run as many adequately-detailed simulations as we would like. We may therefore not have statistically adequate data for the claims we want to make. The problem above compounds this.

8 Conclusions – situation coverage cannot be avoided

Testing that inadequately covers the situations that the AR will encounter is inadequate testing. Explicit measurement of situation coverage therefore has potential to improve our AR testing, and it is practical to implement. There are a great many decisions to be made about how best to do it, and it raises all the normal epistemic challenges of serious use of test coverage criteria.

Acknowledgements

The authors would like to thank Susan Stepney, John Clark, Tim Kelly and Alan Winfield for their contributions to our understanding of this topic, and John McDermid for his specific advice on this report. Alexander and Hawkins are currently funded to work in this area by EPSRC grant EP/L00643X/1.

Glossary

- AoR – Ahead-of-Run
- AR - Autonomous Robot
- IR – In-Run
- SC – Situation Coverage
- SGen – Situation Generation
- SUT – System Under Test – the AR that we're testing (as distinct from any other ARs we have put in the situation as part of the test)

References

[1] S. Dogramadzi, M.E. Giannaccini, C. Harper, M. Sobhani, R. Woodman, J. Choung: "Environmental Hazard Analysis - a Variant of Preliminary Hazard Analysis for Autonomous Mobile Robots". *Journal of Intelligent Robot Systems* (2014)

[2] *MoD Interim Defence Standard 00-56 Issue 4 - Safety Management Requirements for Defence Systems*. Ministry of Defence (2007)

[3] M. Thompson: "Testing the Intelligence of Unmanned Autonomous Systems". *ITEA Journal* **29**, pp. 380-387 (2008)

[4] J.C. Knight, N.G. Leveson: "A Large Scale Experiment In *N*-Version Programming". In: *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pp. 135-139. (1985)

[5] G. Gay, M. Staats, M.W. Whalen, M.P.E. Heimdahl: "Moving the goalposts: coverage satisfaction is not enough". In: *Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST)*, pp. 19-22. ACM, (2014)

[6] M. Grindal, J. Offutt, S.F. Andler: "Combination testing strategies: a survey". *Software: Testing, Verification and Reliability* **15**, pp. 167-199 (2005)

[7] S. Poulding, R. Alexander, J.A. Clark, M.J. Hadley: "The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing". In: *Proceedings of GECCO 2013*, pp. 1477-1484. (2013)

[8] B. Bauer, J.P. Müller, J. Odell: Agent UML: A Formalism for Specifying Multiagent Interaction. In: Ciancarini, P., Wooldridge, M. (eds.) Agent-Oriented Software Engineering, pp. 91-103. Springer (2001)

[9] C. Kaner, J. Bach, B. Pettichord: *Lessons Learned in Software Testing: A Context Driven Approach*. John Wiley & Sons (2002)

[10] J.A. McDermid, T.P. Kelly: "Software in Safety Critical Systems: Achievement and Prediction". *Nuclear Future* **2**, pp. 140-146 (2006)

[11] P.B. Ladkin: "Ontological Analysis". *Safety Systems* **14**, (2005)

[12] J.H. Andrews, L.C. Briand, Y. Labiche: "Is mutation an appropriate tool for testing experiments?". In: *Proceedings of the International Conference on Software Engineering*. (2005)

[13] R. Baker, I. Habli: "An Empirical Evaluation of Mutation Testing For Improving the Test Quality of Safety-Critical Software". *IEEE Transactions on Software Engineering* **39**, (2013)

[14] M. Hadley, J.A. Clark: "Good Days and Bad Days: Investigating Effectiveness and Reliability of “Optimum” Test Sets". In: *Proceedings of Mutation 2013*. (2013)