Verifiable Autonomy
— how can you trust your robots?

Michael Fisher (*and Louise Dennis*)

Department of Computer Science and
Centre for Autonomous Systems Technology
University of Liverpool

Part II: *Verification and Application*

Overview

- *Formal Verification*
  ... what do we mean by "*formal verification*"?
  ... many varieties of formal verification.

- *Brief Introduction to Model Checking*
  ... temporal logics, model-checking, Büchi Automata
  model-checking *programs* and *Java PathFinder (JPF)*

- *Agent Verification and AJPF*

- *Case Studies*
  ... formal verification of UAV decisions
  ... towards verification *ethical decision-making*
  ... etc ...

What is Verification?

Verification [dictionary]:

> *Additional proof that something that was believed (some fact or hypothesis or theory) is correct.*

Verification [of a system]:

> *Establishing that the system under construction conforms to the specified requirements.*

So: we want to carry out *verification* of systems to *show that the system matches its requirements*.

Formal Verification [of a system]:

> *The act of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics.*

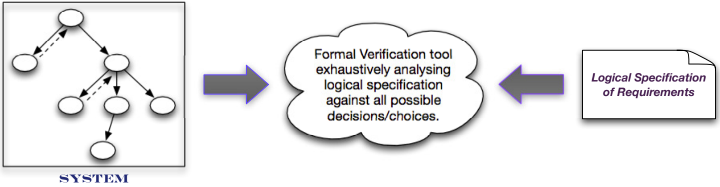
Verifying Logical Requirements

As we have seen, there is a wide range of logical dimensions with relevance to our requirements, such as time, location, uncertainty, context, resources, etc.

Even beyond this, there are a number of different mechanisms for carrying out verification.

Imagine that we have a *formal requirement*, perhaps in the form of a modal/temporal logic formula, R.

This is to be matched against some system we are interested in.



Verification Varieties (1)

Proof: where the behaviour of the system is described by the logical formula, S, and verification involves *proving* $\vdash S \Rightarrow R$.

Typically, this requires (automated) deductive methods able to cope with combinations of logics.

Model-Checking: where R is checked against all possible execution paths within the system.

All these executions are usually described using a finite state structure, typically an automaton such as $\mathcal{A}$.

Our system satisfies R so long as, for every path σ through the automaton $\mathcal{A}$, then we can show that $\sigma \models R$.

Verification Varieties (2)

Dynamic Fault Monitoring (aka Runtime Verification): where the executions *actually* generated by the system are checked against R.

Given a real system execution, σ, then a finite-state automaton representing the property R is used to iteratively scan the execution produced to check that it indeed satisfies R.

Program Model-Checking: where, instead of assessing R against a *model* of the system (e.g. $\mathcal{A}$ above), then R is checked against all *actual* executions.

This depends on being able to generate <u>all</u> the program executions — typically, this requires *symbolic execution*.

$\Rightarrow$ we are particularly concerned with this last variety.

Understanding Model-Checking

The simplest way to explain program model-checking is to start by explaining ‘traditional’ model-checking and work from that.

In turn, the simplest way to explain model-checking is to use *finite automata*.

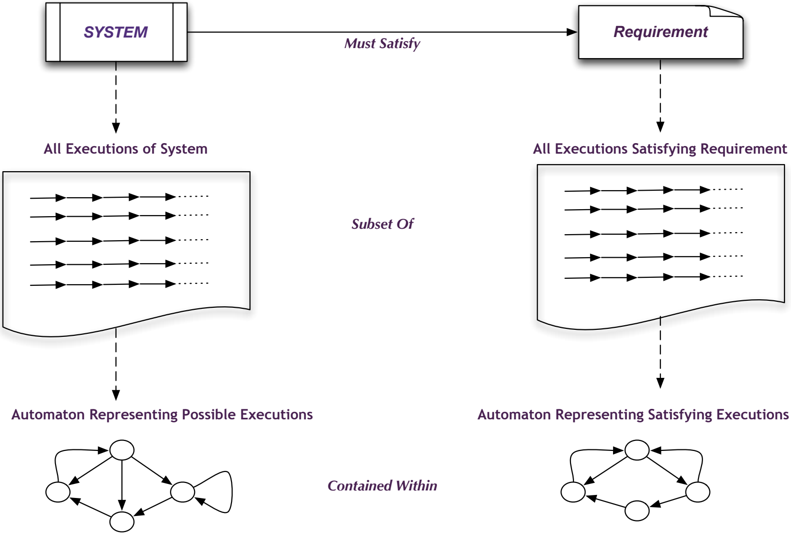
However, the finite automata that we use accept *infinite* strings — they are called Büchi Automata.

The details are not so important, the key aspects being

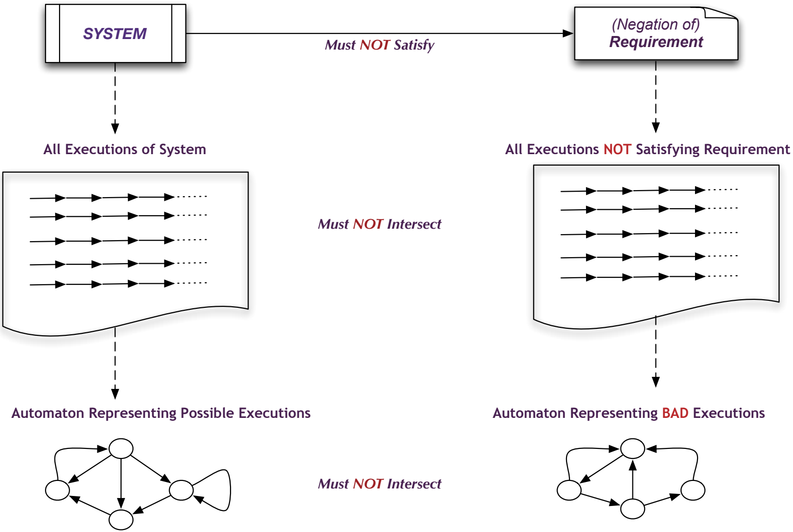
1. they are finite structures, and
2. they represent sets of infinite strings.

These strings will be used to represent both execution sequences and models of logical (typically, temporal/modal logic) formulae.

Automata-Theoretic Model Checking (1)



Automata-Theoretic Model Checking (2)

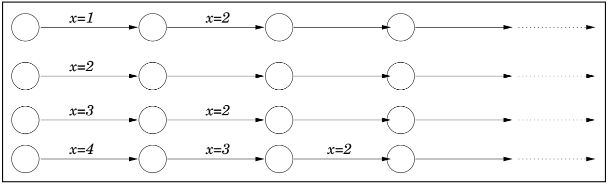


Example Program

```
int x = random(1,4);   /* randomly choose 1, 2, 3 or 4 */

while (x != 2)
do
   if  (x < 2) then x:=x+1; fi
   if  (x > 2) then x:=x-1; fi
od
```

Sample Executions:



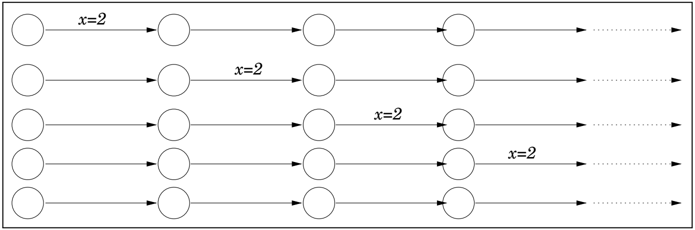
Example Property

Our requirement is that

"At some moment in the future x will have the value 2"

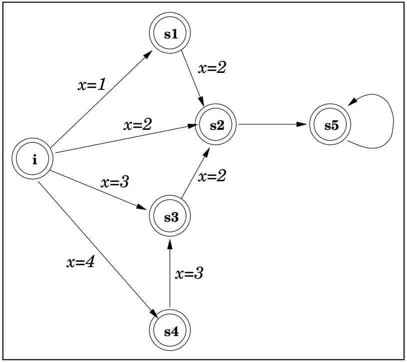
Formal property to check: $\boxed{\Diamond(x = 2)}$

Possible models satisfying this property:

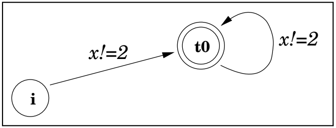


Automata For Example

We construct two automata:



$BA_{program}$



$BA_{\Box(x \neq 2)}$

Note: negation of the $\Diamond(x = 2)$ property is '$\Box(x \neq 2)$'.

Product of Automata

We want to check that

$$sequences\_of(BA_{program}) \cap sequences\_(BA_{\neg\varphi}) \;=\; \emptyset$$

So that: no execution of the program also is a model for $\neg\varphi$

Taking intersections is not so convenient, so we go further, changing the above to a check that

$$sequences\_of(BA_{program} \times BA_{\neg\varphi}) \;=\; \emptyset$$

In other words there is no sequence accepted by the combined automaton; thus, a key aspect of many model checkers is constructing $BA_{program} \times BA_{\neg\varphi}$

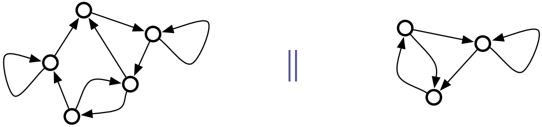
"On the fly" Product Construction

Constructing automata products such as

$$B_S \times B_{\neg\varphi}$$

can be *very* expensive. For example, the number of states in the product automaton may be HUGE.

Rather than combining the two automata explicitly, the "on the fly" approach explores *all* the paths through B_S and, as we do so, simultaneously checks whether any path satisfies $B_{\neg\varphi}$.



Model of the System | *Parallel Exploration* | **Model of "Bad" paths**

Program Model Checking

What do we need in order to be able to implement the on-the-fly model checking approach:

1. a mechanism for extracting *all possible* runs of a system;
2. some way to step the monitoring automaton forwards, as each run proceeds; and
3. a way of recognising *good/bad* looping situations.

Within model-checkers (such as `Spin`) these were achieved by (1) an automaton representing all system executions, (2) a *monitoring* process running synchronously with the main program execution, and (3) an algorithm for recognising Büchi acceptance.

Now that we wish to tackle a high-level language such as `Java` we need these again.

Java Model Checking

The particular approach we consider here is implemented as the `Java PathFinder` system, which is an explicit-state open source model checker for `Java` programs.

The key aspects that allow `Java PathFinder` to achieve:

1. a mechanism for extracting *all possible* runs of a system;
2. some way to step the monitoring automaton forwards, as each run proceeds; and
3. a way of recognising *good/bad* looping situations,

are that

a) it incorporates a modified virtual machine and that
b) *listener* threads are used.

Modified Virtual Machine

Programs in Java are compiled to a set of *bytecodes* which are then executed, when required, by a *virtual machine*, called the Java Virtual Machine (JVM).

In order to allow this execution to be controlled, and indeed backtracked if necessary, Java PathFinder provides a special, modified JVM which explores all executions including all non-deterministic choices, thread interleavings, etc.

Importantly, this new JVM records all the choices made and can backtrack to explore previous choices.

Note that this modified JVM is actually implemented in Java and so runs on top of a standard JVM.

Java Listeners

A `Java` listener is a mechanism within the `Java` language allowing the programmer to "watch" for events.

`Java PathFinder` uses a *listener* in order to provide a representation of an automaton that is attempting to build a model based on the program execution.

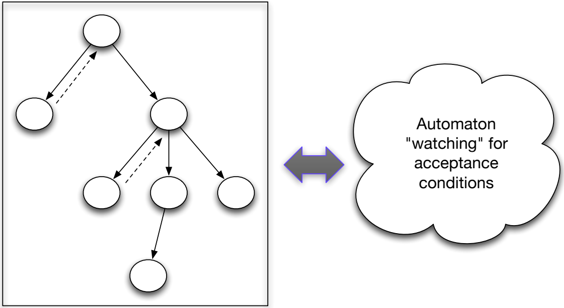
As the program proceeds, the listener recognises state changes in the execution and checks this against its automaton representation.

At certain times the listener may be reset, forcing the `JVM` to backtrack. If the listener recognises an execution sequence, then it reports this.

Since we define the listeners to correspond to "bad" sequences, then the reported sequences are counter-examples.

JPF: Java Verification

A general, pictorial, view of `Java PathFinder` is given below.



Modified Java virtual machine exploring all possible execution branches, not only by forward execution but by backtracking

It combines (backtracking) symbolic execution and a monitoring automaton.

Efficiency (1)

Java PathFinder is now quite well developed and is used for many Java applications.

While extremely useful, Java PathFinder is inherently slow.

It is built upon Java itself so, for example, code that is running executes on the modified JVM, which in turn runs on the standard JVM.

In order to improve efficiency, Java PathFinder employs a variety of sophisticated techniques.

As well as standard partial-order reduction used in many model-checkers, two additional aspects are interesting.

Efficiency (2)

Rather than just exploring the runs through a program in arbitrary order, the user can specify a "*choice generator*" which will explore branches in a specific order.

The other main enhancement involves ensuring that the listener is only forced to move forward if *important* changes occur in the `Java` execution.

Thus, 'unimportant' state changes/operations are collected together into one 'important' state.

The *Model Java Interface* (`MJI`) is a feature of `Java PathFinder` that effectively allows code blocks to be treated as atomic/native methods. Consequently, since new states are *not* built by `Java PathFinder` for calls to atomic/native methods, these code blocks are effectively hidden from the listener.

AJPF

Program model-checking allows us to directly verify the code.

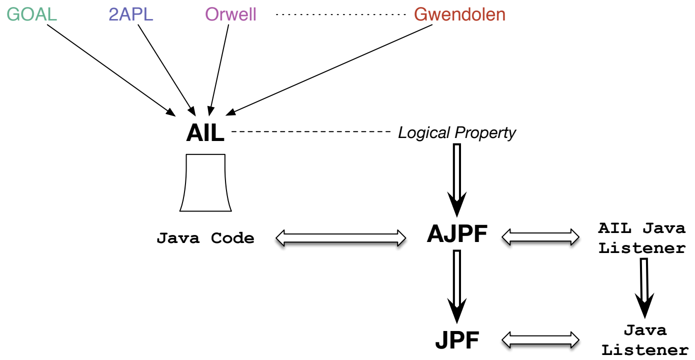
The key aspects that allow `Java PathFinder` to achieve "on the fly" checking are that that it incorporates a modified virtual machine (capable of symbolic execution and backtracking) and that *listener* threads are used (to monitor executions).

Program model checking is *significantly* slower than standard model-checking applied to models of the program execution.

AJPF extends JPF with support for rational agent programs, and a property specification language, based on LTL, and specialised to rational agent programs.

AJPF can model check agent programs and multi-agent systems for any code base which implements the MCAPL interfaces.

AJPF: Anatomy of an Agent Model Checker



AJPF is essentially JPF2 with the theory of AIL *built in*.

The whole verification and programming system is called `MCAPL` and is freely available on Sourceforge:
`sourceforge.net/projects/mcapl`

The Property Specification Language

The Property Specification Language is based on LTL with five "modalities" for agent concepts.

- $B(ag, f)$ (ag believes f)
- $G(ag, f)$ (ag has a goal f)
- $I(ag, f)$ (ag intends to achieve f)
- $P(f)$ (f is perceptible in the environment)
- $A(ag, f)$ (the last action taken was ag doing f)

We have a simple syntax for writing properties:

```
[] (B(searcher, leave)
     -> (B(searcher, found) || B(searcher, area_empty)))
```

The Role of Formal Verification

While *formal verification* techniques have been developed for many aspects of hybrid architectures, e.g. control systems, we choose to instead use formal verification just on the rational agent.

Thus, we verify the system's decision-making, not the real-world outcome of the actions it takes.

Consequently, we verify

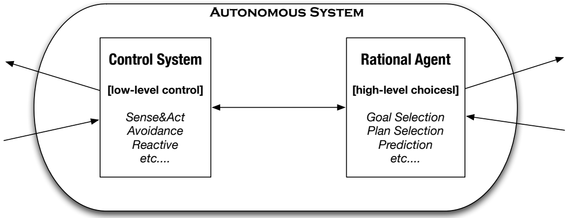
> what the autonomous system chooses to do, given its beliefs

rather than

> what effect the autonomous system has on the world

Separating Decision-Making

So, we separate out the *decision-making* aspect of the system.



The rational agent is typically non-deterministic, but finite.

The 'control' part is typically

1. deterministic, in that it has a predictable feedback interaction with its environment, but
2. potentially infinite, as the environment can be arbitrary.

Verification and Testing

So, we utilise other techniques for verification of the ‘control’ part:

- could use *formal verification* for hybrid systems;
- could use *approximation techniques* for differential equations;
- could use analytical *mathematical proof* if viable; but
- typically use *testing* because of complex environments.

Verifying Autonomous Systems

So, once we have

- an *autonomous system* based on rational agent(s), and
- a *logical requirement*, for example in modal/temporal logic,

We typically use:

- testing to assess the range/correctness of the control part;
- formal verification of the rational agent, possibly with assumptions about the control/environment interactions; and
- simulation of the whole system to give ‘confidence’ to developers.

N.B: Large-scale testing often carried out via HPC.

N.B: Verification of same agent program as used in simulation gives increased confidence.

Verifying UAVs

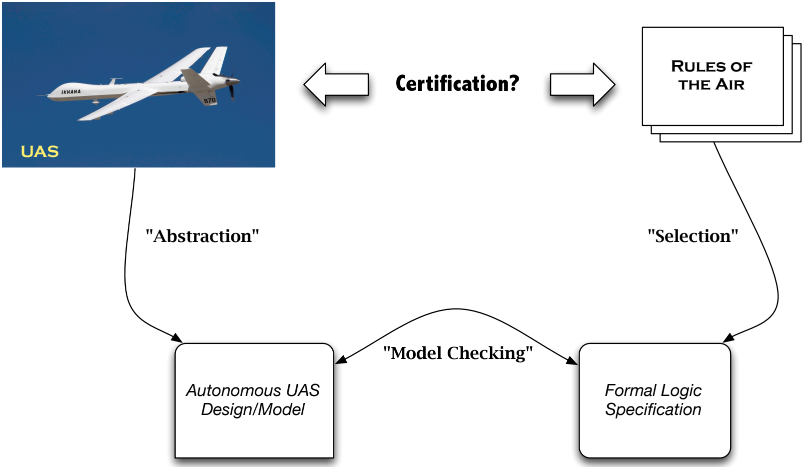
What's the core *difference* between a UAV and a manned aircraft?

Obviously: one uses an "autonomous agent" instead of a pilot!

So, why can't we verify that the "agent" behaves just as a pilot would? i.e. is the agent *equivalent to* the pilot??

This is clearly *impossible*, but......

Our Approach



BDI Agent Controlling the UAV

Our UAV agent has:

- Beliefs, for example
  - waiting at runway
  - turning right (e.g. during *sense & avoid*)

  .....
- Desires, for example
  - complete the mission
  - avoid near-misses

  .....
- Intentions, for example
  - taxi to runway and hold position
  - turn right to avoid object approaching head-on (i.e. *sense & avoid*), for example

  .....

Selected "Rules of the Air"

- *"An aircraft shall not taxi on the apron or the manoeuvring area of an aerodrome without [permission]"*
- *"... when two aircraft are approaching head-on, or approximately so, and there is danger of a collision, each shall alter its course to the right."*
- *"[An aircraft in the vicinity of an aerodrome must] make all turns to the left unless [told otherwise]"*

Note both the ambiguity and the possible conflict!

Verification of Basic UAV Agent

Basic UAV Agent comprises 36 plans, but is relatively straightforward.

It taxis, holds, lines up and takes off, and once airborne it performs simple navigation and sense/avoid actions. Finally, it lands.

Then verify simple properties, e.g "avoidance":

$$\left[\begin{array}{c} B_{\mathsf{uav}}\mathrm{changeHeading} \wedge \\ B_{\mathsf{uav}}\mathrm{nearAerodrome} \wedge \\ \neg B_{\mathsf{uav}}\mathrm{toldOtherwise} \end{array}\right] \quad \Rightarrow \quad \neg B_{\mathsf{uav}}\mathrm{direction}(\mathrm{right})$$

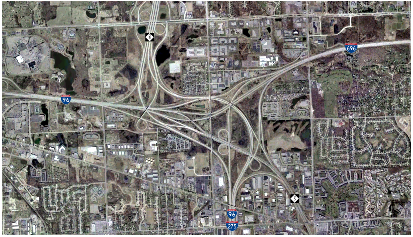
Towards Certification

While clearly not *sufficient* for certifying UAVs, this form of verification is *important* to show that the UAVs does whatever a pilot *should* do.

Of course there is more to a pilot than just following the Rules of the Air ...

From Legality to Ethics

Autonomous systems must make decisions in unexpected situations
→ here some ethical principles are invoked.



UAV has failure → unavoidable crash → but has *some* control

Assesses possible crash sites, but time is running out:

1. on school
2. on field full of animals
3. on a road

Verifying Ethical Decision-Making

System can order options based on *ethical* priorities:

save humans $>>$ save animals $>>$ save property

Once the agent decisions take ethical concerns into account then we can extend formal verification to also assess these.

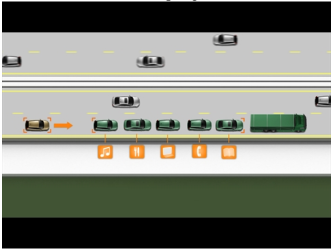
For example, we can formally verify that

if *the chosen course of action violates some substantive ethical concern, A*

then *the other available choices all violated some concern that was equal to, or more severe than, A.*

Road Trains: Safety

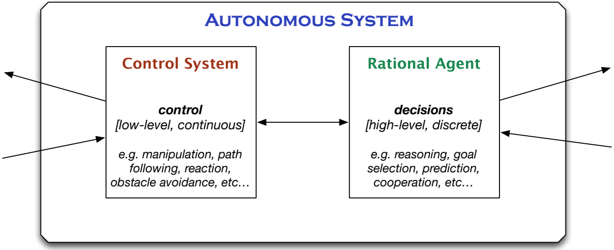
www.sartre-project.eu:



Underlying control system manages distances between vehicles. Rational agent makes decisions about joining/leaving, changing control systems, etc.

Verifying Rational Agent to ensure that convoy operates appropriately.

Verifying Ethical Decision-Making?



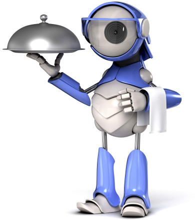
Formal verification of internal rules/plans against pre-determined (safety, legality) criteria.

In unexpected situations, planner invoked and agent must decide between options.

So verify the decision-making approach against the appropriate ethical ordering.

Overview

As long as your autonomous system has an appropriate hybrid agent architecture, then we can use these techniques for verify it:



Note:

1. we have to be able to formally describe requirements
2. we cannot guarantee properties of the environment
3. however, we can guarantee properties of internal/agent software

Issues: Speed

Program model checking is significantly slower than standard model-checking applied to models of the program execution.

It carries out *symbolic execution* of the program.

In addition, “random” environments provide large state spaces.

Thus, verifications in AJPF take minutes and hours, rather than seconds with tools such as Spin or NuSMV.

Work is under way to try to improve this....

Issues: Logics

- Choosing the *best* logics to use to describe our requirements can be difficult.
- Have we *asked* all the right questions?
- Are some properties *too* difficult to describe?
- Specifications such as

$$B_{me}^{>0.75} \Diamond G_{assistant}\, sell\_shoes(me) \;\Rightarrow\; I_{me} \Diamond^{<5s} leave\_shop(me)$$

  require quite complex combinations — we do not yet have full verification systems for these.

\+ N.B: many requirements are logically *very* simple....
... do not require nested modalities!

Concluding Remarks

Key new aspect in Autonomous Systems is that the system is able to *decide for itself* about the best course of action to take.

Rational Agent abstraction represents the core elements of this autonomous decision making:

- (uncertain) *beliefs* about its environment,
- *goals* it wishes wish to achieve and,
- *deliberation* strategies for deciding between options.

Clearly, *formal verification* is needed.

By verifying the rational agent, we verify not *what* system does, but what it *tries* to do and *why* it decided to try!

Thanks to *many* people.....

The work described in this tutorial involved *many* people.....

- Matt Webster (Virtual Engineering Centre, Daresbury)
- Clare Dixon (Computer Science, Univ. Liverpool)
- Rafael Bordini (UFRGS, Brazil)
- Alexei Lisitsa (Computer Science, Univ. Liverpool)
- Sandor Veres (Engineering, Univ. Sheffield)
- Savas Konur (Computer Science, Univ. Sheffield)
- Mike Jump (Engineering, Univ. Liverpool)
- Owen McAree (Engineering, Univ. Sheffield)
- Maryam Kamali (Computer Science, Univ. Liverpool)
- Neil Cameron (Virtual Engineering Centre, Daresbury)
- Nick Lincoln (Engineering, Univ. Southampton)

- EPSRC, for funding many of these activities.

Selected/Related References

- Baier, Katoen. *Principles of Model Checking*. MIT Press, 2008.
- Clarke, Grumberg, Peled. *Model Checking*. MIT Press, 2000.
- Dennis, Fisher, Slavkovik, Webster. Formal Verification of Ethical Choices in Autonomous Systems. To appear in *Robotics and Autonomous Systems*, 2016.
- Dennis, Fisher, Webster. Verifying Autonomous Systems. *Communications of the ACM 56(9):84–93*, 2013
- Dennis, Fisher, Webster. Two-stage Agent Program Verification. *Journal of Logic and Computation*, 2016.
- Dennis, Fisher, Webster, Bordini. Model Checking Agent Programming Languages. To appear in *Journal of Automated Software Engineering 19(1):5-63*, 2012.
- Dennis, Fisher, Lincoln, Lisitsa, Veres. Practical Verification of Decision-Making in Agent-Based Autonomous Systems. *Journal of Automated Software Engineering*, 2016.
- Dennis, Fisher, Winfield. Towards Verifiably Ethical Robot Behaviour. Proc. First International Workshop on AI and Ethics. AAAI, 2015
- Fetzer. Program Verification: The Very Idea. *Communications of the ACM 31(9):1048–1063*, 1988.
- Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*. Wiley, 2011.
- Gerth, Peled, Vardi, Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proc. 15th Workshop on Protocol Specification Testing and Verification (PSTV)*, pp3–18. 1995.
- Havelund, Rosu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.
- Kamali, Dennis, McAree, Fisher, Veres. Verification of Joining and Leaving Protocols for Autonomous Vehicle Platooning.
- Konur, Fisher, Schewe. Combined Model Checking for Temporal, Probabilistic, and Real-time Logics. *Theoretical Computer Science 503:61-88*, 2013.
- Sistla, Vardi, Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.
- Vardi, Wolper. Reasoning About Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.
- Visser, Havelund, Brat, Park, Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003. See *Java PathFinder*: `http://javapathfinder.sourceforge.net`.
- Webster, Cameron, Fisher, and Jump. Generating Certification Evidence for Autonomous Unmanned Aircraft Using Model Checking and Simulation. *J. Aerospace Information Systems 11(5):258–279*, 2014.