Monitoring Robotic Systems using CSP: From Safety Designs to Safety Monitors[*]

Matt Luckcuck
Department of Computer Science, University of Liverpool, UK

8th July 2020

Abstract

Runtime Verification (RV) involves monitoring a system to check if it satisfies or violates a property. It is effective at bridging the *reality gap* between design-time assumptions and run-time environments; which is especially useful for robotic systems, because they operate in the real-world. This paper presents an RV approach that uses a Communicating Sequential Processes (CSP) model, derived from natural-language safety documents, as a runtime monitor. We describe our modelling process and monitoring toolchain, VARANUS. The approach is demonstrated on a teleoperated robotic system, called MASCOT, which enables remote operations inside a nuclear reactor. We show how the safety design documents for the MASCOT system were modelled (including how modelling revealed an underspecification in the document) and evaluate the utility of the VARANUS toolchain. As far as we know, this is the first RV approach to directly use a CSP model. This approach provides traceability of the safety properties from the documentation to the system, a verified monitor for RV, and validation of the safety documents themselves.

1 Introduction

Runtime Verification (RV) techniques check properties of a System Under Analysis (SUA), either by monitoring a running program (online RV) or by checking program logs (offline RV). RV bridges the *reality gap* between design-time assumptions and run-time environments, which is especially useful for robotic systems because they are often safety-critical and operate in real-world environments. Checking that a robotic system remains within safe boundaries is especially important if they interact with humans.

This paper presents an RV approach for safety-critical robotic systems, where the a Communicating Sequential Processes (CSP) model is used as the oracle. The model is built, by hand, from exiting natural-language safety documentation. Our monitoring toolchain uses the CSP model checker, the Failures-Divergences Refinement checker (FDR) [11], to compare the events from the SUA with the model of its safety functions.

This paper describes several contributions. First, the approach validates the safety documents themselves, as the act of formally modelling the safety design can expose oversights in the document(s). This happened during the validation of our model for this work, which is discussed in §6. Second, the approach provides direct traceability of the safety properties described in the safety documentation to a single module in the robotic system. Alongside this, the formalisation and the verification of those properties gives strong evidence that the runtime monitor works correctly and captures the safety properties. This would be beneficial, for example, for demonstrating system safety to a regulator. Finally, this is the first such work directly using a CSP model – though there are approaches using *implementations* of CSP (see §2).

[*]Work supported through the UKRI RAIN grant EP/R026084, and enabled by previous work carried out within the framework of the EUROfusion Consortium, which has received funding from the Euratom research and training programme 2014-2018 and 2019-2020, grant No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission. Thanks are owed to Luigi Pangione and Rob Skilton at RACE; Matt Webster for helpful discussion; and Angelo Ferrando, Louise Dennis, and Marie Farrell for their comments on early drafts.

We illustrate our approach on the MASCOT system (described in §4) which is a pair of master-slave robotic arms that enable engineers to operate remotely inside a fusion reactor. For the upgrade to MASCOT version 6, which includes some autonomous action in the slave arms, a new safety analysis and design were produced. These safety documents are the source material for our formal model and safety properties.

The rest of the paper is laid out as follows; §2 describes related work, and §3 describes our RV toolchain and its intended workflow. In §4 we describe the MASCOT system, its CSP model, and how we validated and verified the model. § 5 describes how we evaluate the toolchain. In §6 we discuss the development of this approach and toolchain. Finally, §7 concludes the paper, summarising our approach and describing future work.

2 Related Work

As far as we know, this work is the first to directly use CSP models for RV. CSP_E [12], is a shallow-embedded Domain Specific Language (DSL) in Scala, specifically for RV. However, CSP_E is missing internal choice ($\sqcap$), τ, and event hiding. Again, this made the conversation from the CSP model of our case study to CSP_E difficult to do systematically.

Some RV approaches have been implemented for Robot Operating System (ROS), which is a prevalent software framework for robotic systems. ROSRV [5] tackles both safety and security RV of ROS programs. For safety monitoring, a specification language is provided to describe the properties to monitor. C++ ROS nodes are automatically generated from the monitor specifications. One drawback is that multiple monitors are centralised into a single (multi-threaded) process. Recently, a similar framework, ROSMon [3], was built to provide updated RV for ROS. ROSMon is able to monitor the messages exchanged between ROS nodes and query an oracle to check if they should be passed on to the destination node or filtered out. But since our case study does not use ROS, these were not applicable approaches.

There are other approaches that are designed to be more generally applicable to robotic systems. For example, [7] uses Computation Tree Logic (CTL) and model checking to produce safety model specifications for robotic system. They partition the states of the robotic system into three categories: "catastrophic states", which violate a safety invariant; "warning states", which can lead to a catastrophic state; and "safe states", which are surrounded by "safety trigger conditions" that prevent transitions from warning to catastrophic states. Their safety invariants are based on a given hazard analysis.

There are also examples of existing work that highlights how RV bridges the reality gap. For example, [4] presents an RV approach that highlights when the environmental assumptions made during design-time formal verification are invalidated by interactions with the real world.

3 Toolchain and Workflow

This section describes our RV toolchain and its intended workflow. Our toolchain, Varanus[1], checks the SUA against a CSP model of its safety system. It does this by sending traces of events performed by the SUA to FDR and returns the verdict, whether the trace passes or fails. This verdict is currently presented to the user, but it could be used by the SUA to avoid or mitigate the violation.

Figure 1 illustrates the Varanus toolchain and workflow. Briefly, we (1) formalise the system's safety properties and functions (extracted from safety documents), then (2) verify that the model satisfies the safety properties – to validate the model against the documentation. Then, (3) Varanus listens for events emitted by the SUA and then (4) checks the events against the model (using FDR) to determine if the system is performing the safety functions.

Varanus[2] is implemented in Python, and is modular to aid adaptation when analysing new systems. It links the SUA to an existing installation of FDR, via the latter's API. Varanus can listen for evens over a socket or WebSocket, or read events from a file (for offline RV). It constructs a trace from the events it hears, uses FDR to check them against the model, and then returns the verdict from FDR. It also produces a log, including the run time, as discussed in §5.

[1]The biological genus of Monitor Lizards.
[2]The toolchain is available at `http://tiny.cc/varanus-iFM`

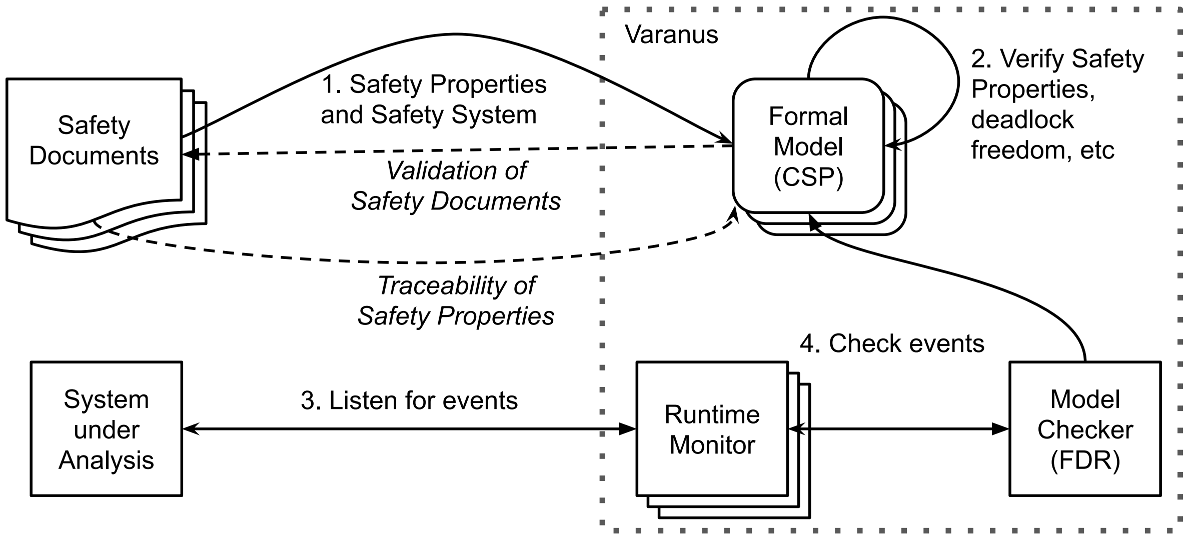
Varanus
Safety Documents
1. Safety Properties and Safety System
Validation of Safety Documents
Traceability of Safety Properties
Formal Model (CSP)
2. Verify Safety Properties, deadlock freedom, etc
System under Analysis
3. Listen for events
Runtime Monitor
4. Check events
Model Checker (FDR)

Figure 1: Diagram of the Varanus toolchain, annotated with the workflow in numerical order. The dashed box shows the Varanus system boundary. The rounded-edge boxes represents the model and the squared-edge boxes represent programs. The arrowheads represent direction of information flow.

The remainder of this section describes the toolchain and workflow. First, § 3.1 describes *Formalising* the safety system and properties from the documentation. §3.2 describes *verifying* the formal model of safety system. §3.3 explains how the toolchain *listens* for events from the SUA and converts them into CSP traces. Finally, §3.4 describes *checking* the events from the SUA against the CSP model, using FDR.

3.1 Step 1: Formalising

This step involves formalising a specification from the system's safety documents, which could take the form of (for example) safety cases or safety designs. A safety cases is a structured argument that the system is (acceptably) safe, which may be natural-language or graphical (for example, Goal Structuring Notation [6] or Structured Assurance Case Metamodel [9]). A safety design is how we refer to a report, or a similar natural-language document, describing how a system will operate safely. This description may take the form of a specific safety sub-system that must obey some safety properties.

If safety cases are available, then their structure may be useful to speed up the extraction of a specification; extracting a specification from a loosely-structured report may be more difficult. In either case, this step involves carefully reading natural-language descriptions and formalising them by hand. This can be time-consuming and requires expertise in formal modelling. This is by no means a novel observation, and discussion in the literature makes it clear that building specifications is still the biggest bottleneck in formal methods [10].

The Varanus toolchain is built to work directly with a CSP model; the model built in this step is used as the RV oracle, producing the verdict on whether a trace from the SUA is valid. In the author's experience, software engineering or safety specifications are often written in a way that enables smooth translation into CSP– this is discussed further in § 6.

The safety properties described in the safety documents are also formalised in CSP, making it easy to verify that the model preserves these properties – as described in §3.2. Further, it provides clear traceability of the properties from the documents into a formal artefact of the system's development.

3.2 Step 2: Verifying

This step takes the CSP model and safety properties extracted from the system's safety documents (described in Sect. 3.1) and verifies that the model preserves the properties. As mentioned in Sect. 3.1, the properties and model are both written in CSP so this step uses the CSP model checker, FDR.

First, the specification must be debugged. This is achieved by using a combination of FDR's built-in assertions (deadlock, determinism, and divergence), and its Probe tool (which allows a user to step through a process's available events) to incrementally identify and remove bugs in the model. Then, the model can be checked against the formalised safety properties. Again, this is an iterative process that may require more specification debugging.

The process of formalising the safety model and verifying that it preserves the safety properties has the side-benefit of validating the safety document itself. The act of formalising the natural-language specification of the safety system's behaviour can highlight inconsistencies in the document. Even simple verification checks, such as checking for deadlock-freedom, can provide counterexamples that show an error in the natural-language specification. We discuss an instance of this that occurred during this work, in §6.

3.3 Step 3: Listening

Once we are confident that the CSP model accurately represents the safety system (and therefore the SUA's behaviour), we can use it for monitoring. Here, part of the Varanus toolchain listens to the SUA and translates the information coming from the SUA into events of the CSP model.

How to attach the monitor and what information that it can receive is highly dependent on the SUA. As we discuss in detail in §4, Varanus can read a log file or receive information via a network. Each SUA is slightly different, so the system interface will need to written with the particular SUA in mind. If the information being sent by the SUA does not match the events in the model, Varanus is capable of using a JSON file to map information from the SUA to the corresponding CSP events. Again, this depends on the specifics of the SUA.

Each event generated by the SUA is adedd to the trace inside Varanus, which begins with the model's initialisation event. New events are appended to the trace. If the SUA is sending events or is producing time-stamped datagrams, then Varanus can produce a single trace. If the updates are more coarse-grained and there is the possibility of receiving a communication where several variables have changed value, with no indication of the order in which these updates happened, then Varanus can split the current trace and produce several *possible* traces, which are then tracked as the SUA continues to run.

3.4 Step 4: Checking

This step checks the CSP trace(s) that have been built from the updates from the SUA (§3.3) against the CSP model. Specifically, Varanus uses FDR's built-in *has trace* assertion to check if the SUA's trace is a valid trace of the model; that is, that the system is behaving according to the model.

Each check is built from the template:

$$\mathit{assert\ MODEL} : [\mathit{has\ trace}] : \langle \mathit{sua\_trace} \rangle$$

where *MODEL* is the CSP process that defines the safety model, and *sua_trace* is a trace of events from the SUA. FDR returns either that the assertion check has passed or failed, and provide a counterexample for a failing result. In our example application(§4), this information is returned to the user. In more complex examples, this information could used by the SUA for replanning or recovery.

As mentioned in §3.3, there might be several traces, depending on the style of updates provided by the SUA. If there is only one trace, then it is sent to FDR and the resultant pass or a fail is, trivially, a 100% result. If there are several traces, then they are all sent to FDR to be checked and the results are expressed as a percentage of the passing traces. In this multi-trace case, if all the traces result in a failure, then the verdict is failure.

4 Case Study: The MASCOT System

This section describes an application of the Varanus toolchain (§3). Our case study is the MASCOT system, which is a pair of master–slave robotic arms used at the Culham Centre for Fusion Energy (CCFE) in the UK to service the Joint European Torus (JET) nuclear fusion reactor. JET is operated by the UK Atomic Energy Authority (UKAEA) under contract from the European Commission, and exploited by the EUROfusion consortium of European Fusion Laboratories[3]. The slave arms mirror the movements of the master arms, which are manually controlled by a human operator, enabling human operators to work remotely inside a nuclear fusion reactor.

[3]`https://www.euro-fusion.org`

Various tools have been adapted to fit on the end of the MASCOT manipulators. Tools can be changed during operations, picked from a ‘tool box’ that is also moved inside the reactor during operations. This has enabled MASCOT to be used used to install, clean, and repair components inside the reactor.

A programme is underway to update the MASCOT system from v.4.5 to v.6, which includes adding an autonomous mode to perform some basic repetitive operations without human intervention. This update has prompted a safety analysis and a new safety design[4] upon which we base our formal model. The key safety concerns relate to keeping the human operator safe during operation and maintenance.

The rest of this section is structured as follows. In §4.1 we describe the pertinent parts of the MASCOT v.6 Safety Design document. § 4.2 describes how the safety design was modelled. Finally, §4.3 describes how the model was validated and verified.

4.1 MASCOT 6 Safety Design

As previously mentioned, the update to MASCOT v.6 introduces an autonomous mode. The safety analysis identifies that when the slave-arms are moving autonomously, the master-arms would mirror this movement, causing a hazard to the operator. Mitigating this risk is the key concern of the safety subsystem described in the safety design, which introduces two modes of operation: hands-on and autonomous, with a different speed limit for the MASCOT arms in each mode.

The safety design defines seven safety ‘concepts’, which are the components of the safety control system. We model six of these components:

1. Emergency/Protective Stop, which controls both manually (Emergency) and automatically (Protective) stopping the system;
2. Safe State Key Switch, which initiates an emergency stop, triggered from either the master or slave cubicle;
3. Master Commissioning Mode Key Switch, which enables the operator to put the master arms into a Commissioning State, for repairs etc.;
4. Slave Commissioning Mode Key Switch, which enables the operator to put the slave arms into a Commissioning State (different to that of the Master arms);
5. Master Safe Speed Monitor, which is responsible for monitoring the speed of the master arms, and raising a Protective Stop if the speed limit is broken; and,
6. Master Hands-on Mode Monitor, which toggles the safety system between the Hands-on and Autonomous modes, in response to the operator using a foot pedal switch.

The final component (which do not model) is an output that indicates the control system is still active, which is indirectly shown by other components in the model.

The Safe Speed Monitor (SSM) and the Hands-On Mode Monitor (HOMM) are the core components of the safety subsystem, cooperating to enforce the speed limit relevant to the current mode. The speed limit when the system in autonomous mode is half that of the speed limit when in hands-on mode.

The HOMM monitors the foot pedal and tells the SSM what mode the subsystem is in. The SSM checks the speed and issues a protective stop if that mode’s speed limit is broken. The other components of the safety system also interact with these core components, adding extra complexity to the system. For example, if the Master Commissioning Mode is activated, then a protective stop is not issued if the speed limit is broken.

Individually, the six components are simple enough to be analysed by hand. However, the interactions between the components make manual analysis of the whole system very difficult. Formalising the components makes the natural-language descriptions into concrete statements that enable automatic checking. In §4.2, we describe the CSP model that corresponds to the MASCOT safety system.

[4]These documents are confidential, so we are unable to make them publicly available.

$$\begin{array}{l}
\mathit{HMM\_AUTONOMOUS\_MODE} = \\
\quad \mathit{enter\_safe\_state} \rightarrow \mathit{HMM\_SAFE\_STATE}(AM) \\
\quad \Box \\
\quad \mathit{foot\_pedal\_pressed.True} \rightarrow \mathit{enter\_hands\_on\_mode} \rightarrow \\
\qquad \quad \mathit{HMM\_HANDS\_ON\_MODE} \\
\quad \Box \\
\quad \mathit{speed}?\_ \rightarrow \mathit{HMM\_PAUSE}(AM) \\
\quad \Box \\
\quad \mathit{enter\_slave\_commissioning\_state} \rightarrow \mathit{HMM\_SAFE\_STATE}(AM)
\end{array}$$

Figure 2: An extract from the *HOMM*, showing the process controlling the *HOMM* in autonomous mode .

4.2 Modelling Approach

This section describes the CSP model that captures the MASCOT v.6 safety subsystem. The model maintains a close correspondence between the components safety system and the model. Four of the six components are each captured as their own process; the remaining two (the Emergency/Protective Stop and the Safe State Key Switch) are represented by a single process, because their behaviour is very closely aligned and their combination produced a simpler model.

The model contains some simplifications to reduce its state space. For example, both the master and salve cubicles will have safe state key switches, but our model abstracts this to one component. Similarly, the speed of three different joints on each of the slave arms should be monitored. Our model abstracts this to one single speed measurement, though since the speed limit for each joint is the same, it would be easy to replicate this process.

Here, we briefly describe the relevant CSP notation. CSP specifications are built from processes. A process may take parameters, and describes a sequence of events; $a \rightarrow b \rightarrow \mathit{Skip}$ is the process where the events a and b happen sequentially, followed by *Skip* which is the terminating process. Processes are connected by bi-directional channels. An event is a communication on a channel, which may accept input ($a?in$), produce output ($b!out$), or require a particular event on that channel ($c.param$). Parameters must match the specified type for that channel. Further, a restricted input can be specified ($d?p : set$) which allows any communications on d that matches the channel's type and are in the *set*. $P \Box Q$ provides the option of either P or Q to the process's environment, once one process is picked the other becomes unavailable. Additionally, processes can occur in sequence or run in parallel.

The safety system is represented by the *MASCOT_SAFETY_SYSTEM* process, which comprises the processes for each of the components operating in parallel. Each process begins with, and must synchronise on, the *system_init* event, which ensures that all of the components start executing at the same time.

The model contains two processes that do not represent safety concepts and are just modelling artefacts. The *MASCOT_SYSTEM_STATE* process, which tracks the state (Safe, Autonomous, Hands-On, and the Master and Slave Commissioning modes) that the system is in – a process was used because CSP does not have variables. We have assumed that these states are mutually exclusive. The *ATOM_CHAINS* process, which enforces certain atomic chains of events that are required by the safety properties – for example, when the foot pedal is pressed, the next event is a change of mode.

The *HOMM* starts in autonomous mode, this is based on the assumption that the operator is not pressing the foot pedal when the system is initialised. Figure 2 shows the process controlling the *HOMM* in autonomous mode. The *foot_pedal_pressed* event toggles the process between hands on mode and autonomous mode. In Fig. 2, *foot_pedal_pressed.True* takes the *HOMM* process into hands on mode (the *HMM_HANDS_ON_MODE*).

Two events, *enter_safe_state* and *enter_slave_commissioning_state*, trigger a change to the safe state, which is controlled by the *HMM_SAFE_STATE* process. This can happen in either mode. In Fig. 2 the *HMM_SAFE_STATE* process is called with the *AM* parameter, which tells it to return to the autonomous mode when leaving the safe state. The process controlling the hands on mode has a

$$
\begin{array}{l}
\mathit{SSM\_AUTONOMOUS\_MODE} = \\
\qquad \mathit{enter\_hands\_on\_mode} \rightarrow \mathit{SSM\_HANDS\_ON\_MODE} \\
\qquad \Box \\
\qquad \mathit{speed}?s : \mathit{AutonomousSafeSpeeds} \rightarrow \mathit{speed\_ok} \rightarrow \\
\qquad\qquad \mathit{SSM\_AUTONOMOUS\_MODE} \\
\qquad \Box \\
\qquad \mathit{speed}?s : \mathit{AutonomousUnSafeSpeeds} \rightarrow \mathit{protective\_stop} \rightarrow \\
\qquad\qquad \mathit{enter\_safe\_state} \rightarrow \mathit{SSM\_SAFE\_STATE}(\mathit{AM}) \\
\qquad \Box \\
\qquad \mathit{enter\_safe\_state} \rightarrow \mathit{SSM\_SAFE\_STATE}(\mathit{AM}) \\
\qquad \Box \\
\qquad \mathit{enter\_slave\_commissioning\_state} \rightarrow \mathit{SSM\_SAFE\_STATE}(\mathit{AM})
\end{array}
$$

Figure 3: An extract from the *SSM*, showing the process controlling the *SSM* in autonomous mode .

similar parameter.

Finally, in either mode, the *speed* event pauses the *HOMM* to ensure that only either a *protective_stop* or a *speed_ok* event can occur. This is handled by the *HMM_PAUSE* process, which in Fig. 2 is called with the *AM* parameter to tell it to return to the autonomous mode when resuming. The *protective_stop* event indicates that the speed limit has been broken, so the *HOMM* enters autonomous mode – regardless of which mode it was in previously. Conversely, *speed_ok* indicates that the speed is within the speed limit, so *HOMM*goes back to its previous mode. Both of these events are driven by the *SAFE_SPEED_MONITOR* (*SSM*).

The *SSM* process also starts in autonomous mode and toggles between that and hands-on mode, but here the trigger to change modes is an event driven by the *HOMM*. This allows the mode change to occur at the same time, but leaves the *HOMM*to handle the *foot_pedal_pressed* event alone. This sort of modular design is repeated throughout the model, where one process handles an external event and communications with other processes via an internal channel.

Figure 3 shows the process controlling the *SSM* in autonomous mode. The *enter_hands_on_mode* event is driven by the *HOMM*, and triggers the *SSM* to change to hands on mode. The events *enter_safe_state* or *enter_slave_commissioning_state* trigger a change to the safe state; similarly to the *HOMM* this calls a process with the *AM* parameter so that the *SSM* returns to autonomous mode when leaving the safe state.

The key purpose of the *SSM* is to handle the *speed* event. In Fig 3, if the speed parameter is in the set of safe speeds for the autonomous mode, $\mathit{speed}?s : \mathit{AutonomousSafeSpeeds}$, then the process responds with *speed_ok*. If the speed parameter is in the set of unsafe speeds for the autonomous mode, $\mathit{speed}?s : \mathit{AutonomousUnSafeSpeeds}$, then the process esponds with *protective_stop* and enters the safe state. These two events drive the *HOMM*, as previously described.

For brevity, we have omitted more detailed discussion of the interactions between the other processes in the model[5]. The next section describes how we validate the model against the safety design.

4.3 Model Validation and Verification

This section describes two steps in validating the model against the safety design and verifying that it preserves the safety properties. This phase ensures that the model correctly represents the safety design, meaning that it can be used for runtime verification.

First, we validate the model, carefully checking that it represents the safety design. We use FDR's built-in assertions, which check for deadlock, divergence, and determinism; and its Probe tool. The assertion checks automatically identify undesirable behaviour in the model. The Probe tool lets us step through a process, choosing the order of events. Both are invaluable tools for specification debugging.

[5]The model is available at `https://zenodo.org/record/3932005`

This is an iterative process where: the model is checked, compared to the safety design, and edited (where needed). This step provides confidence that the model accurately captures the safety concepts.

Specification debugging revealed a problem in the safety design document, which presented as a deadlock between the *SSM* and the *HOMM*. As before, the *SSM* checks the MASCOT arm's speed against the speed limit for the current mode, autonomous or hands-on; while the *HOMM* checks if the foot pedal has been pressed, changing the mode to autonomous if it has.

The deadlock was caused by a conflict between the *HOMM* and *SSM*, where the mode could be changed before the speed had been checked. This allowed the speed limit to be changed, after a potentially unsafe speed had been recorded but before the *protective_stop* could be raised. Discussions with the MASCOT team at Remote Applications in Challenging Environments (RACE) confirmed that this was incorrect behaviour. Removing this bug from the model was relatively simple, once it was confirmed that the *SSM* should take precedence over the *HOMM*. Identifying bugs like this shows the utility of careful specification debugging. We discuss this further in §6.

Next we verified that the model preserves the safety properties, which are the safety requirements of the safety subsystem. This step shows that the model implements the requirements. Again, this is an iterative process, where some properties were found not to hold and this prompted alterations to the model. In turn, this required edits that led back to the validation step.

In CSP, a safety property is specified as a process. The CSP model checker, FDR, checks that the safety specification is refined by the system specification; that is, that the system implements that safety property.

To illustrate the model verification, we use the *HOMM* process as an example. Three statements in Safety Design are requirements and so are captured as safety properties. The simplest is the requirement in point 5.7.1(6) "The monitored foot pedal is the only way for Hands-on Mode to be entered". This is captured by the safety specification:

$$HMM1 = foot\_pedal\_pressed.True \rightarrow enter\_hands\_on\_mode \rightarrow HMM1$$

The $HMM1$ process allows the foot pedal to be pressed ($foot\_pedal\_pressed.True$) and then enters hands on mode ($enter\_hands\_on\_mode$). We use FDR to check that the $MASCOT\_SAFETY\_SYSTEM$ process implements $HMM1$.

The next identified requirement is a little more complicated: *"Autonomous mode is entered if the control system indicates it is no longer in Hands-on Mode"*. This is modelled by the safety specification:

$$\begin{aligned}
&HMM2 = \\
&\quad foot\_pedal\_pressed.False \rightarrow enter\_autonomous\_mode \rightarrow HMM2 \\
&\quad |\sim| \\
&\quad enter\_autonomous\_mode \rightarrow HMM2
\end{aligned}$$

$HMM2$ allows a non-deterministic choice ($|\sim|$) between detecting that the foot pedal has not been pressed, then entering autonomous mode; or entering autonomous mode, which is there to allow the system to perform this event when triggered by something else that doesn't effect this safety specification. This is needed to stop this check failing when another process triggers the change to autonomous mode, for example if the system enters the safe state it changes to autonomous mode. Again, we use FDR to check that the model of the system implements $HMM2$.

Similar safety specifications and checks are performed for the processes modelling the other safety concepts. This provides confidence that the model implements the requirements in the Safety Design. The validation and debugging of the model is also crucial for confidence that it is modelling the safety design correctly, and gives the benefit of checking the safety document 'for free'.

5 Toolchain Evaluation

This section describes how the Varanus toolchain (§3) has been evaluated[6], using the case study presented in §4. As previously mentioned, Varanus can be used for both online (listening for events from a running SUA) or offline (reading a log file) RV. The trace of events observed from the SUA

[6]The log files and results are available at `https://zenodo.org/record/3932005`

Trace Length	FDR (s)	Varanus Offline (s)	Varanus Online
10	0.16	0.11	0.08
100	0.20	0.14	0.10
1000	0.48	0.37	0.27
10000	2.86	2.82	1.90
100000	29.97	30.34	18.31

Table 1: Results, in seconds, for checking traces of length 10, 100, 1000, and 10,000 in FDR and Varanus (offline and online). Note: trace length is exclusive of the *system_init* event that represents the system starting.

is checked against the model, using FDR, and the result is returned to the user. If the trace is valid, then the system is behaving as specified.

MASCOT v.6 is still under development, so it was not possible to test either online or offline RV directly on the MASCOT system, or to compare the execution times of MASCOT with and without monitoring. Instead, we examined Varanus's response times on constructed traces. First, we stress-tested our approach, with increasingly long, semi random traces. Then, we built a set of scenarios that might occur during a hypothetical mission, based on MASCOT log files and personal correspondence with the MASCOT team at RACE. All of the results are from running Python 2.1.17 and FDR 4.2.4 on PC using Ubuntu 19.10, with an Intel Core i5-3470 3.20 GHz $\times$ 4 CPU, and 8 GB of RAM.

Both the stress-testing traces and the scenario traces were checked directly in FDR and using Varanus, both offline and online. For the offline case, Varanus reads the whole trace from a file and sends the trace to FDR to be checked. In the online case, Varanus receives events one-by-one from a small Python script that is sending the events down a TCP socket. Varanus accumulates the trace, adding each new event and then sending the current trace to FDR to be checked. These checks give us an idea of the scalability of the model checking in FDR, and the response times for Varanus checking traces from a log file and over a (local) network.

To stress test our approach, we checked traces of *foot_switch_pressed* and *speed* events, that increased in length from 10 to 100,000 events. A small Python script was built to generate the traces so as to make sure that the *foot_switch_pressed* channel's parameter toggled between *true* and *false*, and the *speed* channel's parameter was always below the autonomous (lower) speed limit. This was to ensure that the entire trace would be checked, instead of a counterexample being generated midway through.

Table 1 shows the results (in seconds) of a *has trace* check directly in FDR, using Varanus offline, and using Varanus online. The Trace Length column shows the number of events in the trace. Each trace begins with the *system_init* event, which represents the system starting. This is not included in the stated trace length, to keep the length a round number. The events in the trace come in pairs, so the number of the other events could not just be reduced by one. (not including the *system_init* event which starts the system). The results are the mean over 10 runs of each trace length in each case (FDR, offline, or online).

As we can see, all of the results for traces of 1000 events or fewer, are below 1s. Once the trace length reaches 100000, the response time starts to become too long for effective online use. We discuss mitigations for this issue in more detail in §6. An interesting observation is that running the check directly in FDR is consistently slower than using Varanus offline, which itself is consistently slower than using Varanus online. This is notable, since we expected that using Varanus would add a time overhead, in comparison to using FDR directly. We will see this repeated for the scenario tests, again we discuss this in §6.

To test the response times of our approach when responding to different scenarios, we checked 13 different traces representing different 'attempts' at a hypothetical mission. The mission is for operators to use the MASCOT system to replace insulating tiles on the inside of the reactor. Based on notes in the log files, this appears to be characterised by groups of repeated actions, such as: removing a group of tiles, installing a group of new tiles, tightening the bolts on a group of tiles, etc. The mission is bookended by collecting and then returning the tools required from the MASCOT tool box, which is also present inside the reactor. The data in the log files shows mainly low velocity values, with some spikes that tend to appear much later on of all the, possibly as the task becomes more difficult.

The constructed scenarios, described in Table 2, exercise different features of the safety system,

No.	Concept(s)	Description
1	1, 5, and 6	Operator stays in hands on mode, speed stays below limit.
2	1, 5, and 6	Operator stays in hands on mode, speed exceeds limit and tries to continue (causes a failure).
2a	2	Instead of the failure in Scenario 2, the system handles the broken speed limit, then resets, restarts, and finishes the mission.
2b	2	Instead of the failure in Scenario 2, the system handles the broken speed limit, the safe state key is removed, to allow minor servicing to the system. Then the key is returned, the system is reset, restarted, and the mission is completed.
3	1, 5, and 6	Operator switches to autonomous mode after collecting tools, speed stays below limit.
4	1, 5, and 6	Operator switches to autonomous mode after collecting tools, speed exceeds limit and tries to continue (causes a failure).
4a	2	Instead of the failure in Scenario 4, the system handles the broken speed limit, then resets, restarts, and finishes the mission.
4b	2	Instead of the failure in Scenario 4, the system handles the broken speed limit, then safe state key is removed, to allow minor servicing to the system. Then the key is returned, the system is reset, restarted, and the mission is completed.
5	2	The Safe State Key is used to trigger an emergency stop. Then the system is reset, restarted, and the mission is completed.
6	3	System enters Master Commissioning Mode. After some unmonitored movements (not triggering protective stop), Safe State Key is used to enter Safe State, and system is reset.
7	4	The Slave Commissioning Mode key is used to put the system into the Slave Commissioning Mode, where no speed events are registered. Then Slave Commissioning Mode is disabled, again using the Slave Commissioning Mode key.

Table 2: Table of scenarios used to evaluate the Varanus toolchain, showing the Scenario Number, the identifier of Safety Concepts that it tests, and its description.

under a variety of circumstances. They test different combinations of events, mixing changes between autonomous and hands on modes, different speeds, and other safety system components. Between them, the scenarios cover the all of the safety concepts modelled from the safety design.

Table 3 shows the results (in seconds) of running a *has trace* check directly in FDR, using Varanus offline, and online. The Trace Length column shows the number of events in the trace (which now includes the *system_init* event). As with Table 1, the results are the mean over 10 runs of each trace length in each case (FDR, offline, or online). Scenarios 2 and 4 are built to fail, so while they are both 193 events long they fail after 84 events and 66 events, respectively.

Again, with these results we see that the time taken to directly check the traces in FDR is always slower than checking using Varanus Offline, which itself is always slower than checking using Varanus Online. Again, this is interesting because Varanus uses the same FDR installation as the direct checks. This is discussed in detail in §6

Given the results of the stress-testing (Table 1) it is no surprise that all of the scenario traces were checked in less than 1s, since the traces are all fewer than 1000 events. It is likely that these trace lengths aren't indicative of the length of the traces that would be generated by the real system, again mitigation strategies for this are discussed in §6. That being said, event the longest traces (4a, 200 events; 4b, 203 events; and 5, 201 events) are all checked quickly: 0.24s or 0.25s in FDR, 0.17s with Varanus Offline, and 0.12s with Varanus Online.

The range of trace lengths for scenarios 1–5 is 48 (and only 10 if we exclude scenarios 2a and 2b) and the range between the results is 0.05 for checking in FDR, 0.03 using Varanus Offline, and 0.01 Varanus Online. This suggests that the response time is most dependent on trace length as there is little discernable difference in the time taken to check a trace that exercises one safety concept (such as scenarios 5 or 6) and those exercising multiple safety concepts (such as scenarios 1, 2, 3, and 4). A more detailed study of FDR is required to answer deeper questions about how long more complex traces or assertions might take to check, this is left for future work.

Scenario Name	Trace Length	FDR (s)	Varanus Offline (s)	Varanus Online
Scenario 1	193	0.26	0.16	0.12
Scenario 2	193 (84)	0.23	0.15	0.12
Scenario 2a	155	0.24	0.16	0.11
Scenario 2b	158	0.22	0.15	0.11
Scenario 3	193	0.24	0.16	0.12
Scenario 4	193 (66)	0.21	0.14	0.12
Scenario 4a	200	0.25	0.17	0.12
Scenario 4b	203	0.24	0.17	0.12
Scenario 5	201	0.25	0.17	0.12
Scenario 6	45	0.18	0.12	0.09
Scenario 7	10	0.17	0.11	0.08

Table 3: Results, in seconds, for checking traces of the scenarios in Table 2, in FDR and Varanus both offline and online. Each result is a mean over 10 checks. Note Scenarios 2 and 4 are built to fail, the length of the trace up to the failure is noted in brackets.

6 Discussion

RV has several benefits. First, it helps to mitigate the *reality gap*, which is the problem faced when trying to transfer models (particularly of a system's environment) to the real world because of uncertainty, abstractions, and simplifications to the model. Trying to improve the fidelity of formal models to reduce the reality gap, often makes the model intractable [2]. RV partially side-steps this problem by monitoring the system at run-time and comparing its behaviour to that expected by the model. This can be used to enforce safety claims and check assumptions made at design-time.

Another benefit of RV, particularly in our approach, is that it provides traceability of the requirements into a system component. In our case, this is the model itself, and the links between the requirements and parts of the model have been discussed in §4.3. Further, Varanus checks the formal model directly, which obviates the need for a monitor to be implemented or synthesised from the model. This improves our confidence that the monitor is checking the correct properties, because they have not been implemented and distributed throughout a program.

Another benefit of both clear traceability and of RV itself is the part the both play in a safety case. Varanus checks at runtime that the MASCOT safety system works as described in the safety design. This provides another line of defence against system failures. Clear traceability makes it easier to show where in the system this checking is implemented, so it can be compared to the requirements.

The process of modelling the safety system is also beneficial. Ideally the safety system model would be built before the safety system is implemented, to provide a (formal) specification for the system. Even if the model is built before the system, the model provides an independently-built (formal) 'implementation' of the requirements against which the system can be checked. It is a common adage that specification is the biggest bottleneck in using formal methods [10], so making the most out of the effort by using a specification for implementation, verification, and RV seems advantageous.

CSP was chosen for this work because, in the author's experience, it enables the relatively easy modelling of software systems specifications. Earlier incarnations of the work described in this paper attempted to convert our model into existing *implementations* of CSP; two in Python (Python-CSP [8] and PyCSP [1]) and the other in Scala (CSP_E [12]). However, all three implementations differed slightly from 'pure' CSP, which made systematic conversion difficult.

The modelling process also has the side-benefit of checking the safety documentation, for 'free'. Formalising requirements forces ambiguous natural-language descriptions to be clarified, and checking a model for things like deadlock-freedom tests the system before it is implemented.

As previously mentioned, we found a problem in the MASCOT 6 safety design where two concurrent components could produce a spurious deadlock. The safety design didn't specify which of these components should take precedence, this was known but omitted. Model checking highlighted this subtle error, because it automatically and exhaustively explores the model's state space. After a discussion with a member of the MASCOT team at RACE, the model was carefully updated to fix the bug. As we highlighted in §5, this highlights the importance of specification debugging.

The results in both Tables 1 and 3, surprisingly, show that checking *has trace* directly in FDR is consistently slower than using VARANUS Offline, and that using it Offline is consistently slower than using it Online. The direct FDR checks used the GUI, whereas VARANUS uses FDR's API. Perhaps calculating some extra information for visualisation that is used only by the GUI caused the response time. It is important to note that even if direct checking in FDR were quicker it does not provide the automatic preparation and checking of traces that VARANUS performs.

The even more surprising result is that using VARANUS Online is faster than using it Offline. For the online case, VARANUS starts an FDR session and sends traces that increase in length by 1 event, each time. FDR seems to be caching some information from previous checks (probably its representation of the model) which helps to reduce the time of subsequent checks. In the future, we may modify VARANUS 's offline process to read and check events one-by-one, to take advantage of this time reduction. This optimisation in FDR is helpful, but the stress tests (Table 1) show the upper limits of this part of the toolchain, where traces of 100001 events still take 18.31s to check.

The response times of all of the traces, except the two largest stress-testing traces, are less than 1s in all three cases. The largest of these traces 203 events long, but the MASCOT logs contain many thousands of records. It is unclear how quickly MASCOT v.6 will produce events, but it is likely that the trace will eventually become intractably long. In this case we can see two paths for optimisation. (1) traces could either be filtered, so they only contain events for one safety concept; sampled, to produce shorter representative traces; or reset after the SUA completes a cycle of behaviour that takes it back to the initial state. These ideas would need careful study to prevent mistakes. (2) investigate optimisations in FDR, or else replace it with a bespoke evaluation approach. Studying both of these avenues is left for future work.

7 Conclusion

This paper presents a novel RV framework for monitoring a running program against a CSP model. The events from the SUA are converted into a CSP trace and then FDR is used to test if the model accepts the trace. This is implemented in a toolchain called VARANUS.

As far as we know, this is the first RV framework that operates directly on CSP models. CSP is designed for specifying processes, so it is suited to capturing imperative descriptions of behaviour. In the author's experience, it is relatively easy to use CSP to model systems described for or by software or mechanical engineers

We demonstrated the approach, and the VARANUS toolchain, using the MASCOT teleoperation system as an example. We formalised the safety system and safety properties, described in an English-language safety design document. We stress-tested VARANUS to find the response times for traces of different lengths. This showed that traces over 1000 events might not be checkable online. Since the MASCOT system does not currently implement the safety system, we evaluated VARANUS on a set of scenarios constructed from MASCOT log files and personal communication with members of the MASCOT team at RACE.

For future work, we intend to apply the approach to safety cases and examine what benefits can be gained when the source material for the safety properties is more structured. If possible, we would like to automate the extraction of safety properties from a safety case – or at least highlight the likely nodes in safety case that contain the safety properties to ease the workload on the modeller. We also intend to improve VARANUS, to provide better user feedback after a failing trace is found, and to better test its ability to monitor running systems. Finally, we intend to investigate ways to improve VARANUS's online response times.

References

[1] John Markus Bjørndalen, Brian Vinter, and Otto J. Anshus. Pycsp - communicating sequential processes for python. In *Communicating Process Architectures Conference*, volume 65 of *Concurrent Systems Engineering*, pages 229–248, 2007.

[2] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. Combining Model Checking and Runtime Verification for Safe Robotics. In *Runtime Verif.*, volume 10548 of *LNCS*, pages 172–189, 2017.

[3] Angelo Ferrando. ROSMonitoring: a Runtime Verification Framework for ROS, 2020.

[4] Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. Recognising Assumption Violations in Autonomous Systems Verificaion. In *Autonomous Agents and Multiagent Systems*, pages 1933–1935. IFAAMAS/ACM, 2018.

[5] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. ROSRV: Runtime Verification for Robots. In *Runtime Verification*, volume 8734 of *LNCS*, pages 247–254. Springer, 2014.

[6] Tim Kelly and Rob Weaver. The Goal Structuring Notation – A Safety Argument Notation. In *Dependable Systems and Networks: Workshop on Assurance Cases*, 2004.

[7] Mathilde Machin, Fanny Dufossé, Jean-paul Blanquart, Jérémie Guiochet, David Powell, and Hélène Waeselynck. Specifying Safety Monitors for Autonomous Systems Using Model-Checking. In *Computer Safety, Reliability, and Security*, volume 8666 of *LNCS*, pages 262–277. Springer, 2014.

[8] Sarah Mount, Mohammad Hammoudeh, Sam Wilson, and Robert Newman. CSP as a domain-specific language embedded in python and jython. 67:293–309, 2009.

[9] Object Management Group. Structured Assurance Case Metamodel (SACM) Version 2.0. Technical Report formal/2018-02-02, March 2018.

[10] Kristin Y Rozier. Specification: The Biggest Bottleneck in Formal Methods and Autonomy. In *Verified Software. Theories, Tools, and Experiments*, volume 9971 of *LNCS*, pages 8–26. Springer, 2016.

[11] Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[12] Yoriyuki Yamagata, Cyrille Artho, Masami Hagiya, Jun Inoue, Lei Ma, Yoshinori Tanabe, and Mitsuharu Yamamoto. Runtime Monitoring for Concurrent Systems. In *Runtime Verification*, volume 10012 of *LNCS*, pages 386–403. Springer, 2016.