On the cloud-enabled refinement checking of railway signalling interlockings

Andrew Simpson and Jaco Jacobs

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD,
United Kingdom

Abstract. Railway signalling systems have received a great deal of attention from the formal methods community. One reason for this is that the domain is relatively accessible; another is that the safety analyses to be undertaken are often highly parallelizable. In this paper we describe a 'cloud interface' for the refinement checker, Failures Divergences Refinement (FDR), which has been motivated and validated by an approach to the modelling and analysis of railway signalling interlockings. In particular, the approach allows us to perform safety checks on systems consisting of billions of states.

1 Introduction

Railway signalling systems have received a great deal of attention from the formal methods community. Early contributions include those of Hansen [1], Morley [2], and Haxthausen and Peleska [3]. More recent contributions include those of Kanso *et al.* [4], James and Roggenbach [5], and Haxthausen *et al.* [6]. In many ways, this level of attention is unsurprising. Crucially, the domain is relatively accessible, enabling researchers to comprehend the problem at hand, and communicate their intentions and solutions to a receptive audience. Another reason, which is offered by Fantechi *et al.* [7], is the fact that the safety-criticality of the domain is attractive to formal methods researchers. The body of work is substantial: one only has to consider the FMERail contributions from the late 1990s;[1] the fact that such applications are considered a success story for the formal methods community (see, for example, Bacherini *et al.* [8]); and the upcoming 2013 Workshop on a Formal Methods Body of Knowledge for Railway Control and Safety Systems.[2]

We would argue that another reason for this relative success is that the safety analyses that can be undertaken are — depending on the model and the approach used — often parallelizable. To this end, decomposition approaches have been proposed by Simpson *et al.* [9], Winter and Robinson [10], as well as others.

In this paper we revisit the contribution of [9] — which utilised Communicating Sequential Processes (CSP) [11, 12] and the associated refinement checker

[1] See `http://www2.imm.dtu.dk/ dibj/fmerail/fmerail/`.
[2] See `http://ssfmgroup.wordpress.com/`.

Failures Divergences Refinement (FDR) [13, 14] — as a means of motivating and validating a cloud-enabled approach to refinement checking. Specifically, we utilise the open source Eucalyptus framework [15] to demonstrate how the (mostly) theoretical decomposition approach described [9] can now be made practical — enabling the checking of systems consisting of billions of states in a matter of minutes.

The structure of the remainder of this paper is as follows. In Section 2 we provide a necessarily brief introduction to CSP and FDR, as well as our case study. Then, in Section 3, we discuss our cloud-enabled interface for FDR. We present our case study in Section 4. Finally, in Section 5, we summarise our contribution, and outline our plans for future work.

2 On CSP, FDR, and railway interlockings

2.1 CSP

The language of CSP is a notation for describing the behaviour of concurrently-evolving objects, or *processes*, in terms of their interaction with their environment. This interaction is modelled in terms of *events*: abstract, instantaneous, synchronisations that may be shared between several processes. We denote the set of all events within a given context as Σ; we can also give consideration to the *alphabet* of a process — the events that a particular process can perform. In the following we introduce a subset of the language of CSP.

We use compound events to represent communication. The event name $c.x$ may represent the communication of a value x on a *channel* named c. At the event level, no distinction is made between *input* and *output*: the willingness to engage in a variety of similar events — the readiness to accept input — is modelled at the process level; the same is true of output, which corresponds to an insistence upon a particular event from a range of possibilities.

A *process* describes the pattern of availability of certain events. The prefix process $e \rightarrow P$ is ready to engage in event e; should this event occur, the subsequent behaviour is that of P, which must itself be a process.

An external (or deterministic) choice of processes $P \mathrel{\Box} Q$ is resolved through interaction with the environment — the first event to occur will determine the subsequent behaviour. If this event was possible for only one of the two alternatives, then the choice will go on to behave as that process. If it was possible for both, then the choice becomes non-deterministic. This form of choice exists in an indexed form: $\Box\, i : I \bullet P(i)$ is an external choice between processes $P(i)$, where i ranges over the (finite) indexing set I.

We may denote input in one of two ways. The process $c?x \rightarrow P$ is willing initially to accept any value (of the appropriate type) on channel c. Alternatively, if we wish to restrict the set of possible input values to a subset of the type associated with the channel c, then we may write $\Box\, x : X \bullet c.x \rightarrow P$.

There are various flavours of parallel combinations; in this paper, we shall limit ourselves to only two. We write $P \parallel Q$ to denote that the component processes P and Q cooperate upon the events appearing in the alphabets of both P and Q, with other events occurring independently. We write $P \mid\mid\mid Q$ to represent the *interleaved* parallel combination of P and Q.

2.2 FDR

The standard notion of refinement for CSP processes, which is defined in [16], is based upon the failures/divergences model of CSP. In this model, each process is associated with a set of behaviours: tuples of sequences and sets that record the occurrence and availability of events.

The *traces* of a process P, denoted $traces\,[P]$, are finite sequences of events in which that process may participate *in that order*; the *failures* of P, denoted $failures\,[P]$, are pairs of the form (tr, X), such that tr is a trace of P and X is a set of events which may be refused by P after the trace tr has been observed. We shall not concern ourselves with divergences.

We write $P \sqsubseteq_M Q$ when Q refines P under the model M — Q is 'at least as good as' P. With respect to failures, the formal definition is as follows:

$$P \sqsubseteq_F Q \Leftrightarrow traces\,[\![Q]\!] \subseteq traces\,[\![P]\!] \land failures\,[\![Q]\!] \subseteq failures\,[\![P]\!]$$

The refinement checker FDR — which utilises the machine-readable dialect of CSP, CSP$_M$ [17] — uses this theory of refinement to investigate whether a potential design meets its specification. A pleasing feature of FDR is that if such a test fails, a counter-example is returned to indicate why this is so.

2.3 Solid State Interlocking

Given the safety-critical nature of railway interlockings, it is important to be able to guarantee a range of safety properties, and, considering the size of the problem, any automated assistance that may support this activity is desirable. The complexity of the task is characterised by Ferrari *et al.* thus:

> "It is a well known fact that interlocking systems, due to their inherent complexity related to the high number of variables involved, are not amenable to automatic verification, typically incurring the state space explosion problem." [18]

Following [9], we consider *Solid State Interlocking* (SSI), a computer-based control system, the system software of which can be divided into generic and specific components. The latter (our concern) varies between locations and describes the signalling functions for that particular instance. We shall use the simple junction of Figure 1 to illustrate a manageable (but still meaningful) subset of the components of interest.

The track is divided into segments by *track circuits*, with each circuit being fitted with a detection device that informs the interlocking if a specific segment is *occupied* (o) or *clear* (c). *Points* help trains navigate junctions and can be either *controlled normal* (cn) or *controlled reverse* (cr). As an example, if a train is

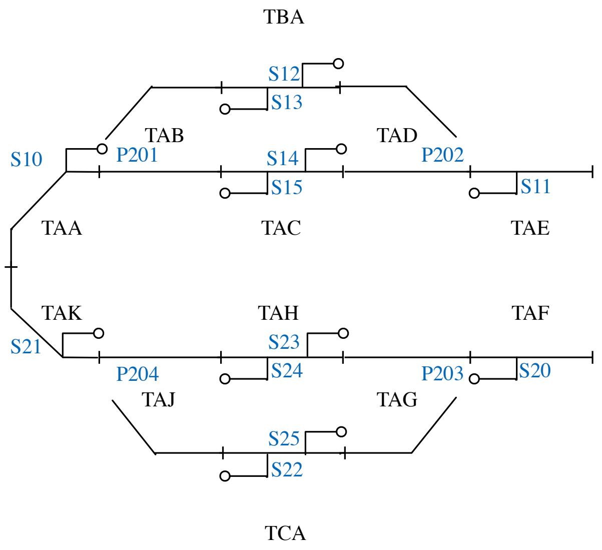

Fig. 1. The Open Alvey interlocking

travelling over track circuit TAK towards track circuit TAJ and points P204 are in controlled reverse position, then the train will follow the section of track covering track circuit TCA. Conversely, if points P204 are in the controlled normal position, the train will continue along track circuit TAJ towards TAH. Boolean checks may be performed on a set of points: these checks indicate whether it is free to move into the controlled normal (*free to go normal*) or controlled reverse (*free to go reverse*) directions. A set of points is *controlled free to go normal* (cfn) if it is free to go normal or if it is already in controlled normal; a set of points is *controlled free to go reverse* (cfr) if it is free to go reverse or if it is already in controlled reverse. A *signal* grants a requesting train entry onto the particular section of track that is under its control. Signal S11, for example, is concerned with track circuits TAD, TAC and TBA. A *route* is a section of track between two signals: route R13 is the section of track between the entry signal S13 and the exit signal S21, running over three track circuits (TAB, TAA and TAK) and one set of points (P201). A route can be *requested* (req), *set* (s), or *unset* (xs). *Subroutes* are sections of routes associated with track circuits; there may exist several subroutes over a particular track circuit. Track circuit TAB, for example, has three entry / exit points (TAA, TAC and TBA), which are labelled clockwise from a 12:00 position. Entry (or exit) from (or to) circuit TBA is labelled A, entry (exit) from (to) TAC is labelled B, and C is associated with entry (exit) from (to) TAA. Subroute UAB_AC is associated with track circuit TAB, with entry from track circuit TBA and exit at track circuit TAA. A subroute can either be *locked* (l) or *free* (f).

The *Geographic Data Language* (GDL) is used for the purposes of describing these interlocking functions in terms of signals, routes, points, etc. We restrict ourselves to a subset of GDL and consider two types of conditional checks: pertaining to route setting data and subroute release data, respectively.

As an example, route R14 runs from signal S14 over track circuits TAD and TAE and points P202. The condition for setting this route is written

O14 **if** P202 cfn UAE_AB f UAD_AB f
then R14 s P202 cn UAD_BA l UAE_BA l

This tests if points P202 are controlled free to go normal and subroutes UAE_AB and UAD_AB are free. If these checks evaluate to true, the route can be granted: points P202 are set to controlled normal, and subroutes UAD_BA and UAE_BA are locked.

Our second type of conditional check pertains to subroutes becoming free. Consider again route R14. When this route is set, subroutes UAD_BA and UAE_BA are both locked. The condition for releasing UAE_BA is written

UAE_BA f **if** TAE c UAD_BA f UAD_CA f

Here, UAE_BA becomes free when track circuit TAE is clear and subroutes UAD_BA and UAD_CA are both free.

There are variations on this pattern. For example, for UAD_BA to become free, track circuit TAD must be clear and route R14 must be unset:

UAD_BA f **if** TAD c R14 xs

In [19] a number of safety invariants for GDL representations are listed. Examples include:

1. If a route is set, then all of its subroutes are locked.
2. For every track circuit, at most one of subroutes passing over it should be locked for a route at any time.
3. If a subroute over a track circuit containing points is locked for a route, then the points are correctly aligned with that subroute.
4. If a track circuit containing points are occupied, then the points are locked.
5. If a subroute is locked for a route, then all subroutes ahead of it on that route is also locked.

In [9, 19] an approach to the modelling, decomposition and analysis of GDL representations is described. By taking advantage of the relationship that exists between refinement and process composition in the failures model of CSP, it is shown how safety checks of potentially billions of states might be decomposed into thousands of checks of hundreds of states — giving rise to a parallelized refinement-checking process. In the following, we show how that largely theoretical process might be made practical via a cloud-enabled version of FDR.

3 A cloud-enabled FDR

3.1 Eucalyptus

Cloud computing — an aggregate of multi-core, multi-processor, distributed compute nodes — enables access to a range of configurable and reliable computing resources that can scale on demand, which, from an automated verification perspective, is extremely desirable. The nature of such activity is bursty: large quantities of computing resources, particularly memory and processing power, are required only when checks are being executed. It follows that the notion of having significant quantities of resources available 'on demand' sits comfortably with automated verification: it provides a viable approach to alleviate the state space explosion problem and has the potential to increase throughput. The notion of computing resources as a utility that can be provisioned and relinquished as needed is a powerful one: it creates the illusion of infinite computing resources, available on-demand, with no prior commitment as to how long they are used. Moreover, when the computing resources are no longer required, they can be released without incurring any penalties.

Cloud computing provision is typically characterised as one of *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS). The first of these is our concern, with the core idea being that computing resources should behave like physical hardware. Users select, control and configure a virtualised server, consisting of the operating system kernel, plus all required libraries, applications and data; administrative tasks (such as provisioning and releasing a virtual server) are typically automated. By having computing instances at such a low level we place few limitations on the software that can ultimately be deployed in this context; a consequence is that it is intrinsically harder for the cloud provider to offer automatic scalability and failover.

Eucalyptus is an open source cloud computing platform that provides an API for provisioning, managing and relinquishing virtual machines in an IaaS cloud; each virtual machine is an *instance*. A virtual machine runs on top of a *hypervisor*, which provides the capabilities necessary in order to provide an isolated computing environment. A user of the cloud has no control over the actual physical server that the instance is run on: it is therefore possible (and highly likely), that the physical hardware that the instance is run on is shared with another instance; (termed *multi-tenancy*). When a user wishes to start a new instance in the Eucalyptus cloud, they do so using a pre-defined machine image, which includes the operating system and any other pre-built software required. It is possible to customise these, create a new image, and then launch the instance using the custom image; this is a *Eucalyptus Machine Image* (EMI).

Eucalyptus is composed of several components that interact through well-defined interfaces. The architecture is modular, with each high-level system component operating as a stand-alone web service that can start, control, access and terminate entire virtual machines using an emulation of the Amazon Elastic Cloud Compute SOAP interface. These components are: *node controllers* (NC in Figure 2), which control VM-related activities (termination, launch, etc.) on

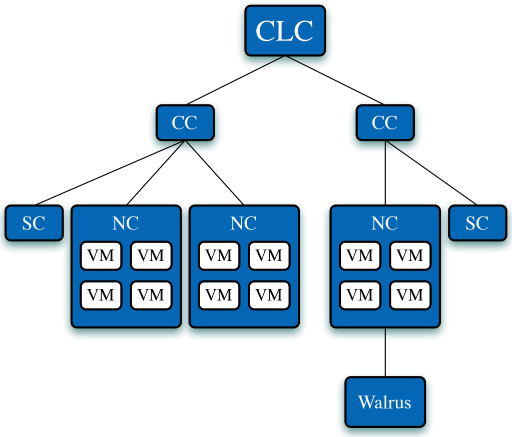

Fig. 2. An example set-up of a Eucalyptus cloud

a compute node; *cluster controllers* (CC), which manage the node controllers within their clusters; *storage controllers* (SC), which can be attached to an instance file system as a volume; *Walrus*, a storage service that provides a mechanism for cloud-based persistent storage; and the *cloud controller* (CLC), which coordinates and manages the cloud as a whole.

Figure 2 illustrates a possible configuration of a Eucalyptus cloud: the single cloud controller communicates with the two cluster controllers, which, in turn, manage the node controllers inside their respective clusters. The node controllers are responsible for executing actions on the physical resources that host virtual machine instances, such as launching, monitoring and shutting down instances.

3.2 A cloud interface for FDR

Parallel model checking techniques typically partition the state space. Our approach involves partitioning the problem not at the level of the state space, but at the level of a particular model. Conceptually, then, we have a CSP model, with a requirement being that the model is such that it allows for checks (expressed as refinements) to be broken down into several, smaller refinements.[3] Once this partitioning is achieved, the refinement checks can then be allocated to a farm of processors to be either confirmed or refuted.

Thus, our process (illustrated in Figure 3) is as follows.

1. Take as input a text file containing a CSP problem description.
2. Automatically derive process definitions from the input file.
3. Automatically extract appropriate process definitions and generate refinement checks by composing the process definitions relevant to the particular refinement check.

[3] We acknowledge that this limits the applicability of the approach.

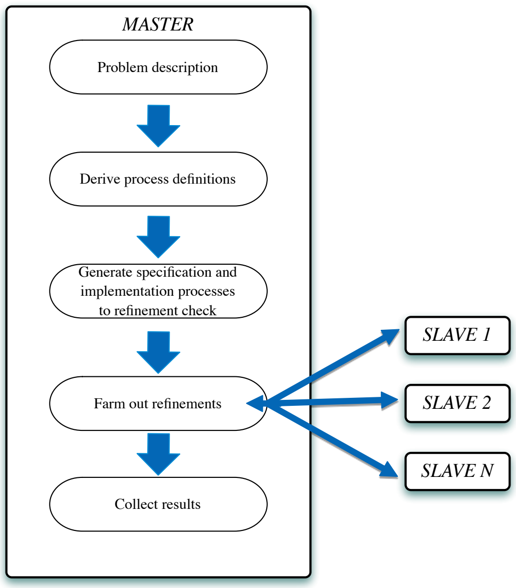
MASTER
Problem description
Derive process definitions
Generate specification and implementation processes to refinement check
Farm out refinements
Collect results
SLAVE 1
SLAVE 2
SLAVE N

Fig. 3. The approach

4. Distribute the refinement checks to compute nodes (each running a server version of FDR).
5. Collect the results and display the end result.

Our case study is characteristic of a problem that can be decomposed into several refinement checks and then distributed to various processing nodes: input to the model checker is a text file representing data for a particular railway interlocking; the CSP model is then automatically derived (along with refinement checks) to assert various safety conditions. These checks can then be distributed to the various processing nodes.

Eucalyptus is used to provide the private infrastructure as a service cloud.[4] The set-up of Figure 4 consists of two servers: the first is configured as the cloud controller, cluster controller, Walrus and storage controller; the second is configured as a node controller capable of booting virtual instances. The node controllers host the virtual instances which boots the machine image containing the FDR binary. Sitting above FDR is the software used to coordinate the scheduling of refinement checks and processing of results. We utilise a single master node and several slave nodes. The role of the master node is to distribute refinements to, and collect results from, slaves. Additionally, the master node is responsible for processing the input file, deriving suitable process definitions, and then extracting the relevant processes in order to form refinements; these are then distributed to the slave nodes.

[4] We use the Ubuntu Enterprise Cloud which uses KVM as the default hypervisor.

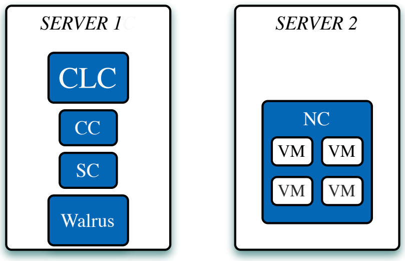

Fig. 4. Eucalyptus set-up

A *job* consists of the relevant CSP code and a refinement to check; jobs are stored in a *jobqueue*. The available pool of slaves are stored in a *slavepool* — a circular list of slaves, keeping a record of whether the slave has been allocated a job. The master node cycles through the list of slaves in a round-robin fashion. If a slave has been previously allocated a job, it checks whether the job is complete. If it is, the result is saved and the slave's state is marked as *idle*; if it is not, the slave is simply added to the back of the list, to be checked on the next cycle. Alternatively, if a slave is free and there are jobs in the job queue, the slave is allocated the next available job, and its state is set to busy. A slave node simply waits for a job from the master. Additionally, it responds to periodic status requests (from the master) as to whether a refinement check is complete or not.

Four types of data are of interest: *application data* (the binary of the model checker, and any other associated applications or scripts); *input data* (CSP$_M$ scripts describing concurrent interactions of processes along with refinements we wish to prove or refute); *non-persistent application-generated data* (data required only for as long as the CSP$_M$ script is loaded and a refinement check is executed); and *persistent application-generated data* (the result of a refinement check and (possibly) counter-examples).

4 The case study

We have used the approach of Section 3 to model various interlockings; as a means of illustration, we use the model of [19] and the example of Figure 1.

4.1 The CSP model

Initially, the system starts in a safe state: all track circuits are clear; all points are controlled normal; no points are free to move in either direction; all subroutes are initially locked; and all routes are initially unset.

The interlocking components involved in setting route R14 are subroutes UAE_AB, UAD_AB, UAD_BA and UAE_BA and points P202. The process $R14true$ characterises when it is possible to set route R14: points P202 are controlled free to go normal, and subroutes UAD_AB and UAE_AB are free. If any of the conditions necessary to set the route becomes false, then the process state is updated and the process subsequently behaves as $R14\mathit{false}$. Should there be a request to set the route, points P202 are locked in the controlled normal position, UAD_BA and UAE_BA are both locked, and route R14 is set.

$$
\begin{array}{l}
R14\mathit{true} = \\
\mathit{routeState}.R14.\mathit{req} \rightarrow \mathit{pointPosition}.P202.\mathit{cn} \rightarrow \\
\qquad \mathit{subrouteState}.\mathit{UAD\_BA}.l \rightarrow \mathit{subrouteState}.\mathit{UAE\_BA}.l \rightarrow \\
\qquad\qquad \mathit{routeState}.R14.s \rightarrow R14\mathit{true} \\
\Box \\
\mathit{pointState}.P202.\mathit{cfn}.\mathit{false} \rightarrow R14\mathit{false}(\mathit{false}, f, f) \\
\Box \\
\mathit{subrouteState}.\mathit{UAE\_AB}.l \rightarrow R14\mathit{false}(\mathit{true}, l, f) \\
\Box \\
\mathit{subrouteState}.\mathit{UAD\_AB}.l \rightarrow R14\mathit{false}(\mathit{true}, f, l)
\end{array}
$$

The process $R14\mathit{false}$ models when it is not possible to set route R14, i.e. when one or more of the conditional checks evaluates to false. The variable x represents the state of points P202 (controlled free to go normal or not); y and z are concerned with the states of subroutes UAE_AB and UAD_AB (free or locked). Changes in state for P202, UAE_AB and UAD_AB may be observed. Once all conditions required for setting the route are met, the process behaves as $R14\mathit{true}$.

$$
\begin{array}{l}
R14\mathit{false}(x, y, z) = \\
\text{if } x = \mathit{true} \wedge y = f \wedge z = f \text{ then } R14\mathit{true} \\
\text{else } (\, \mathit{pointState}.P202.\mathit{cfn}?i \rightarrow R14\mathit{false}(i, y, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\mathit{UAE\_AB}?i \rightarrow R14\mathit{false}(x, i, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\mathit{UAD\_AB}?i \rightarrow R14\mathit{false}(x, y, i) \,)
\end{array}
$$

Subroute-releasing processes are defined similarly. In $\mathit{UAE\_BAlocked}$, variable x represents the state of track circuit TAE, and variables y and z represent the states of UAD_BA and UAD_CA respectively. If all the conditions are met, the subroute can be freed and the process then behaves as $\mathit{UAE\_BAfree}$. Alternatively, the process allows changes to the relevant components, updating the relevant variable accordingly.

$$
\begin{array}{l}
\mathit{UAE\_BAlocked}(x, y, z) = \\
\text{if } x = c \wedge y = f \wedge z = f \text{ then} \\
\qquad \mathit{subrouteState}.\mathit{UAE\_BA}.f \rightarrow \mathit{UAE\_BAfree}(x, y, z) \\
\text{else } (\, \mathit{circuitState}.\mathit{TAE}?i \rightarrow \mathit{UAE\_BAlocked}(i, y, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\mathit{UAD\_BA}?i \rightarrow \mathit{UAE\_BAlocked}(x, i, z) \\
\qquad \Box \\
\qquad \mathit{subrouteState}.\mathit{UAD\_CA}?i \rightarrow \mathit{UAE\_BAlocked}(x, y, i) \,)
\end{array}
$$

The conditional check on subroute UAE_BA when it is free is modelled by the process $\mathit{UAE\_BAfree}$.

$$
\begin{aligned}
&\mathit{UAE\_BAfree}(x, y, z) = \\
&\mathit{subrouteState}.\mathit{UAE\_BA}.l \rightarrow \mathit{UAE\_BAlocked}(x, y, z) \\
&\Box \\
&\mathit{circuitState}.\mathit{TAE}?i \rightarrow \mathit{UAE\_BAfree}(i, y, z) \\
&\Box \\
&\mathit{subrouteState}.\mathit{UAD\_BA}?i \rightarrow \mathit{UAE\_BAfree}(x, i, z) \\
&\Box \\
&\mathit{subrouteState}.\mathit{UAD\_CA}?i \rightarrow \mathit{UAE\_BAfree}(x, y, i)
\end{aligned}
$$

Subroute release data depending on a route rather than subroutes (which is usually the case for the first subroute of a route) are modelled slightly differently. For example, in the case of subroute $\mathit{UAD\_BA}$ we have the following:

$$
\begin{aligned}
&\mathit{UAD\_BAlocked}(x, y) = \\
&\text{if } x = c \wedge y = xs \text{ then} \\
&\qquad \mathit{subrouteState}.\mathit{UAD\_BA}.f \rightarrow \mathit{UAD\_BAfree}(x, y) \\
&\text{else } (\ \mathit{circuitState}.\mathit{TAD}?i \rightarrow \mathit{UAD\_BAlocked}(i, y) \\
&\qquad \Box \\
&\qquad \Box\ i : \{\mathit{req}, \mathit{xs}\} \bullet \mathit{routeState}.R14.i \rightarrow \mathit{UAD\_BAlocked}(x, i)\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{UAD\_BAfree}(x, y) = \\
&\mathit{subrouteState}.\mathit{UAD\_BA}.l \rightarrow \mathit{UAD\_BAlocked}(x, y) \\
&\Box \\
&\mathit{circuitState}.\mathit{TAD}?i \rightarrow \mathit{UAD\_BAfree}(i, y) \\
&\Box \\
&\Box\ i : \{\mathit{req}, \mathit{xs}\} \bullet \mathit{routeState}.R14.i \rightarrow \mathit{UAD\_BAfree}(x, i)
\end{aligned}
$$

4.2 Translating GDL into CSP

We have used the lexical analyser and parser generator *PLY* (a lex–yacc parsing tool for Python),[5] with a representation of the syntax of GDL being given in terms of EBNF.

During the parsing phase, we record semantic information regarding the GDL: this is used to construct process definitions and to decide which processes need to be combined for a particular refinement check. In particular, we record: the set of track circuits, *Circuit*; the set of points, *Points*; the set of routes, *Route*; and the set of subroutes, *Subroute*. In addition, we build a syntax tree that relates the various interlocking components; we also construct various functions that relate different interlocking components. For example, the following

[5] See `http://www.dabeaz.com/ply`.

functions relate track circuits to the subroutes associated with them, and return the set of all locked points for a given route (when the route is set) respectively.

$$\begin{array}{l} \mathit{subroutesOfCircuit} : \mathit{Circuit} \rightarrow \mathbb{P}\ \mathit{Subroute} \\ \mathit{pointsOfRoute} : \mathit{Route} \rightarrow \mathbb{P}\ \mathit{Points} \end{array}$$

The function *subroutesOfRoute* maps a route to its constituent subroutes:.

$$\mathit{subroutesOfRoute} : \mathit{Route} \rightarrow \mathit{seq}\ \mathit{Subroute}$$

The translation tool reads the whole file and then translates it, which involves building tree structures which are efficient at retrieving the components of the file read. Once all the input is parsed we can then transform this into corresponding CSP process definitions.

4.3 Safety invariants in CSP

We now demonstrate how we can model safety invariants. We illustrate this via the first of our invariants: *if a route is set, then all of its subroutes are locked.*

For any route r, we define

$$U = \{u : \mathit{Subroute} \mid u \in \mathit{set}(\mathit{subroutesOfRoute}(r))\}$$

where *set* converts a sequence into a set.

We represent the invariant as a process thus:

$$\begin{array}{ll} \multicolumn{2}{l}{S_1(r, U, \mathit{locked}) =} \\ \multicolumn{2}{l}{\text{if } \mathit{locked} = U \text{ then}} \\ & \Box\, u : \mathit{locked} \bullet \mathit{subrouteState}.u.f \rightarrow S_1(r, U, \mathit{locked} \setminus \{u\}) \\ & \Box \\ & \Box\, \mathit{routeState}.r.s \rightarrow \mathit{routeState}.r.xs \rightarrow S_1(r, U, \mathit{locked}) \\ \multicolumn{2}{l}{\text{else}} \\ & \Box\, u : U \setminus \mathit{locked} \bullet \mathit{subrouteState}.u.l \rightarrow S_{(}r, U, \mathit{locked} \cup \{u\}) \\ & \Box \\ & \Box\, u : \mathit{locked} \bullet \mathit{subrouteState}.u.f \rightarrow S_1(r, U, \mathit{locked} \setminus \{u\}) \end{array}$$

It is clear that we can only set a route r when all the subroutes along that route are locked; we also require the route to become unset before any associated subroutes can become free.

The next step is to derive a suitable implementation process, which involves extracting relevant process descriptions from the GDL and then combining them using parallel composition. The following determines the necessary processes to be composed for $r \in \mathit{Route}$.

1. Include the processes representing route setting data for r.
2. Consider all processes related to subroute release data for each subroute along r, i.e. for each element in the set $\mathit{set}(\mathit{subroutesOfRoute}(r))$.

3. The process $\mathit{Train}(r, \mathit{subroutesOfRoute}(r), \mathit{pointsOfRoute}(r))$ models a train moving along route r.

Consider route $R10A$, where

$$\mathit{set}\,(\mathit{subroutesOfRoute}\,(R10A)) = \{\mathit{UAB\_CA}, \mathit{UBA\_BA}\}$$

It follows that we have

$$I_1\,(R10A) = \\ R10A \parallel \mathit{UAB\_CA} \parallel \mathit{UBA\_BA} \parallel \mathit{Train}(R10A, \langle \mathit{TAB}, \mathit{TBA}\rangle, \{P201\})$$

as the implementation process for safety invariant 1 and route $R10A$. Via FDR we can verify

$$S_1\,(R10A, \{\mathit{UAB\_CA}, \mathit{UBA\_BA}\}, \{\mathit{UAB\_CA}, \mathit{UBA\_BA}\}) \sqsubseteq_F I_1\,(R10A)$$

By verifying similar refinements for all routes in the interlocking, we can assert that safety invariant 1 holds for that interlocking. The proof of this relies on the fact that all relevant behaviours relevant to the verification of the safety invariant for route r can be observed in the implementation process $I_1\,(r)$ (see [19]).

The round-trip execution times for checking each of the 16 routes of Figure 1 are typically in the range 3–5 seconds; this results in a cumulative time of under 1 minute to check this safety invariant for the example interlocking, which consists of $4.84662992 \times 10^{22}$ states;[6] the cumulative times for the other safety invariants are of a similar order.

5 Conclusions

We have described how a cloud-enabled interface for FDR gives rise to a means of parallelized safety checks on railway interlockings. For the sake of readability, we have based our account on a relatively simple scenario; [19] shows how the theoretical approach — which we have made practical — is scalable to 'real-life' interlockings.

One of the biggest challenges of model checking in a practical setting is handling the enumeration of the state space in an efficient manner. Various approaches to alleviate the state space explosion problem are known from the literature: partial order reduction techniques (see, for example, [20]) are one approach; the local search approach proposed by Roscoe *et al.* [21], whereby states spaces are partitioned into 'blocks', is another. An experimental parallel implementation of FDR is described in [22]: states are randomly allocated between different computing nodes using a hash function; the state space is explored using a breadth-first search algorithm, and at the end of each level successor states are

[6] 12 track circuits, 4 points, 16 routes and 30 subroutes, giving rise to $2^{12} \times 4^4 \times 3^{16} \times 2^{30}$ states.

exchanged between the compute nodes. An alternative approach is that taken by FDR Explorer [23], whereby an API "makes possible to create optimised CSP code to perform refinement checks that are more space or time efficient, enabling the analysis of more complex and data-intensive specifications." Our approach involves partitioning the problem not at the level of the state space, but at the level of the CSP model — which means it is applicable only in certain scenarios, with one being the contribution of [19]. All of the refinement checks are generated automatically and subsequently sent to slave nodes for processing.

The initial prototype implementation of the software that schedules the checks between processing nodes can be extended in several ways. At the moment, there is a single point of failure: should the master node die there would be no way to schedule more refinement checks or to collect the results. Another point to consider would be the costing model used by the cloud provider: given that virtual instances are priced per hour, if many of the refinement checks are similar (like in the case study of this paper), we can try and optimise the cost by considering the execution time of a single check. The most pressing item of future work, however, is the consideration of further case studies — with a view to identifying classes of problems that may benefit from this approach.

References

1. Hansen, K.M.: Validation of a railway interlocking model. In Naftalin, M., Denvir, T., Bertran, M., eds.: Proceedings of the 2nd International Symposium of Formal Methods Europe (FME 1994). Springer-Verlag Lecture Notes in Computer Science volume 873 (1994) 582–601
2. Morley, M.J.: Safety in railway signalling data: a behavioural analysis. In Joyce, J., Seger, C., eds.: Proceedings of the 6th Annual Workshop on Higher Order Logic and its Applications, Springer-Verlag Lecture Notes in Computer Science volume 780 (1994) 465–474
3. Haxthausen, A.E., Peleska, J.: Formal development and verification of a distributed railway control system. IEEE Transaction on Software Engineering **26**(8) (2000) 687–701
4. Kanso, K., Moller, F., Setzer, A.: Automated verification of signalling principles in railway interlocking systems. Electronic Notes in Theoretical Computer Science (250) (2009) 19–31
5. James, P., Roggenbach, M.: Automatically verifying railway interlockings using SAT-based model checking. In: Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010), Electronic Communication of the European Association of Software Science and Technology volume 35 (2010)
6. Haxthausen, A.E., Peleska, J., Kinder, S.: A formal approach for the construction and verification of railway control systems. Formal Aspects of Computing **23**(2) (2011) 191–219
7. Fantechi, A., Fokkink, W., Morzenti, A.: Some trends in formal methods applications to railway signalling. In Gnesi, S., Margaria, T., eds.: Formal Methods for Industrial Critical Systems: A Survey of Applications. John Wiley & Sons (2013) 63–82

8. Bacherini, S., Fantechi, A., Tempestini, M., Zingoni, N.: A story about formal methods adoption by a railway signaling manfacturer. In Misra, J., Nipkow, T., Sekerinski, E., eds.: Proceedings of the 14th International Symposium on Formal Methods (FM 2006). Springer-Verlag Lecture Notes in Computer Science volume 4085 (2006) 179–189
9. Simpson, A.C., Woodcock, J.C.P., Davies, J.W.M.: The mechanical verification of Solid State Interlocking geographic data. In Groves, L., Reeves, S., eds.: Proceedings of Formal Methods Pacific 1997. Springer-Verlag (1997) 223–242
10. Winter, K., Robinson, N.J.: Modelling large interlocking systems and model checking small ones. In Oudshoorn, M., ed.: Proceedings of the 26th Australasian Computer Science Conference (ACSC 2003), Australian Computer Science Communications volume 16 309–316
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
12. Roscoe, A.W.: Understanding Concurrent Systems. Springer-Verlag (2010)
13. Roscoe, A.W.: Model checking CSP. In Roscoe, A.W., ed.: A Classical Mind: Essays in honour of C. A. R. Hoare. Prentice-Hall (1994)
14. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M.H., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical compression for model-checking csp or how to check 10^{20} dining philosophers for deadlock. In: Proceedings of the First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1995). Springer-Verlag Lecture Notes in Computer Science volume 1019 (1995) 133–152
15. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus open-source cloud-computing system. In: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2009). (2009) 124–131
16. Brookes, S.D., Roscoe, A.W.: An improved failures model for communicating processes. In Brookes, S.D., Roscoe, A.W., Winskel, G., eds.: Proceedings of the NSF-SERC Seminar on Concurrency. Springer-Verlag Lecture Notes in Computer Science volume 197 (1985) 281–305
17. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall International (1997)
18. Ferrari, A., Magnani, G., Grasso, D., Fantechi, A.: Model checking interlocking control tables. In: Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems 2010 (FORMS/FORMAT 2010). Springer (2011) 107–115
19. Simpson, A.C.: Safety through security. DPhil thesis, Oxford University Computing Laboratory (1996)
20. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag (1996)
21. Roscoe, A.W., Armstrong, P.J., Pragyesh: Local search in model checking. In: Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA 2009), Springer-Verlag Lecture Notes in Computer Science volume 5799 (2009) 22–38
22. Goldsmith, M.H., Martin, J.M.R.: The parallelisation of FDR. In: Proceedings of Workshop on Parallel and Distributed Model Checking (PDMC 2002). (2002)
23. Freitas, L., Woodcock, J.C.P.: FDR Explorer. Formal Aspects of Computing **21**(1–2) (2009) 133–154