Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 19–35
www.elsevier.com/locate/entcs

An Automated Approach for the Interpretation of Counter-Examples

Lionel van den Berg[1,2] Paul Strooper[3] Wendy Johnston[4]

School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, Australia

Abstract

Model checking is an automatic technique used for the verification of finite systems. A model checker explores the full state space of a given model and checks it against a set of requirements. If a state exists in which a requirement is not satisfied most tools will generate a counter-example. Counter-examples are useful for debugging a model and determining if an error exists in the modelled system. However, they can be difficult for end users to understand and this may limit the take-up of model checking in industry. This paper describes a domain-specific approach to automatically interpreting counter-examples and presenting the results in an intuitive form to the end user. Our research extends previous work on model checking railway signalling control tables with signalling engineers from Queensland Rail.

Keywords: Model checking, counter-example, interpretation, railway signalling, verification, animation.

1 Introduction

Model checking is particularly interesting for application in industry, especially to support current processes for verifying safety-critical systems. Model checkers perform an exhaustive search of the full state space of a given model, testing it against a set of properties. Most tools either provide the user with an answer stating that all properties are true, or report a violation in the form of a counter-example [4]. A counter-example is one possible path describing the values of system variables starting at an initial state, and showing each transition required to reach the state in which a property is violated. Model checking is a useful technique for the verification of system designs, however, there are also a number of problems to

[1] Thanks to colleagues Kirsten Winter and Peter Robinson and our associates at Queensland Rail: George Nikandros, David Barney and David Tombs. This project is funded by an ARC Linkage grant LP0455155

[2] Email: lionel@itee.uq.edu.au

[3] Email: pstroop@itee.uq.edu.au

[4] Email: wendy@itee.uq.edu.au

1571-0661
doi:10.1016/j.entcs.2006.12.027

be overcome to achieve acceptance in practice. Counter-examples are difficult for end users to understand [9], particularly in large systems that can have hundreds of variables and in which the user may have little or no knowledge about how the model works.

Previous research between the University of Queensland and Queensland Rail (QR) produced the *Signalling Design Toolset* [12] which includes a track layout editor, a control table generator, a control table editor, and in its preliminary stages a verification manager [14]. One of the control tables produced by the *Signalling Design Toolset* is the route table. The generation of the route table has been formerly verified [13] and is used as a source of data along with the track layout, that is trusted to be correct by the verification manager.

The verification manager is an interface to the NuSMV model checker [2]. It gathers information about the positions of points, signals and tracks from the track layout, and the specification to lock points and routes from the signal control table. Models are built automatically in the NuSMV input language, from these specifications. The properties to be verified are generated based on QR's *Signalling Principles* [11] and are expressed in generic terms as invariants such as avoidance of train collisions and derailment. The verification manager invokes NuSMV which produces a counter-example if a property is violated.

Counter-examples produced by NuSMV typically consist of a first state that includes the values of all variables in the model, and a sequence of subsequent states which include only those variables that have changed since the previous state. When using invariants, the system property being verified is always violated in the last state of the counter-example. However, this is not necessarily where the error occurred in the system specification [8]. Counter-examples are excellent for debugging purposes [4], but they can also be difficult for users to understand. Some problems with counter-examples and model checking include:

- Variables in each state do not always have any relationship to the property that is violated.
- Counter-examples can have a lot of states. Often many of the states are irrelevant to the error in the system specification and are only needed for the model checker to show the full path to the error state.
- The state explosion problem [4] makes it necessary to abstract models in order to reduce the state space [3]. Abstracting a model may mean some data is excluded or modified, and the behaviour of the model can vary from the system specification.

These issues make counter-examples difficult for users to interpret. Our counter-examples are further complicated because the redundancy in the specification of railway interlockings requires us to create models that deliberately leave out some data in order to check other entries in the specification. This causes model behaviour to vary from a real railway interlocking.

Our counter-examples are linear in shape (see Table 1 for an example of a counter-example produced using a small interlocking). They provide a good trace

Trace Type: Counterexample	move_CR = 1	request = reqC_6
-> State: 1.1 <-	request = req_6_1S	routelock_8_1M = rtN
pos_CR = 1Cb	routelock_5_1M = rtR	-> State: 1.9 <-
pos_FS = 1Cb	-> State: 1.3 <-	pos_CR = 2Ca
incoming_train = FS	pos_CR = 1Ca	pos_FS = 8Cb
move_CR = 0	incoming_train = FS	move_FS = 0
move_FS = 0	request = req_8_1M	request = reqN_501
pointState_511 = setR	routelock_6_1S = rtR	routelock_6_1S = rtN
pointState_500 = setR	-> State: 1.4 <-	-> State: 1.10 <-
pointState_501 = setR	pos_CR = 5Aa	pos_CR = 2Ba
request = req_5_1M	incoming_train = CR	pointState_501 = setN
currentRoute_CR = NoRoute	move_FS = 1	request = req_6_2S
currentRoute_FS = NoRoute	request = reqC_5	-> State: 1.11 <-
routelock_5_1M = rtN	currentRoute_CR = 5_1M	pos_CR = 2Aa
routelock_6_1S = rtN	routelock_8_1M = rtR	move_CR = 0
routelock_6_2S = rtN	-> State: 1.5 <-	request = req_4_1M
routelock_7_1M = rtN	pos_CR = 8Bc	routelock_6_2S = rtR
routelock_8_1M = rtN	incoming_train = FS	-> State: 1.12 <-
routelock_8_2M = rtN	request = reqC_6	move_FS = 1
routelock_1_1M = rtN	routelock_5_1M = rtN	request = reqC_6
routelock_1_2M = rtN	-> State: 1.6 <-	routelock_4_1M = rtR
routelock_2_1M = rtN	pos_CR = 8Aa	-> State: 1.13 <-
routelock_3_1S = rtN	pos_FS = 5Ab	pos_FS = 1Bd
routelock_3_2S = rtN	request = reqN_511	request = reqC_4
routelock_4_1M = rtN	-> State: 1.7 <-	currentRoute_FS = 4_1M
routelock_5_1M = rtN	pos_CR = 7Aa	-> State: 1.14 <-
routelock_6_1S = rtN	pos_FS = 8Ab	pos_FS = 1Ab
routelock_6_2S = rtN	pointState_511 = setN	request = req_3_2S
routelock_7_1M = rtN	request = reqC_8	routelock_4_1M = rtN
routelock_8_1M = rtN	currentRoute_FS = 8_1M	-> State: 1.15 <-
routelock_8_2M = rtN	-> State: 1.8 <-	pos_FS = 2Ab
-> State: 1.2 <-	pos_CR = 7Ba	request = reqC_7
incoming_train = CR	pos_FS = 8Bd	routelock_3_2S = rtR

Table 1
A counter-example for a small interlocking.

of train movement and interlocking behaviour up to where the invariant is violated. However, we still found that signalling engineers have difficulty understanding the counter-examples produced by NuSMV for our models. An alternative output to that produced by NuSMV was required. We investigated two approaches for interpreting counter-examples - animation and natural language interpretation. Animation is a highly effective technique. However, in our application animating the railway interlocking and train behaviour from the counter-examples had a number of problems and was not economical. Interpretation is a simpler approach and requires minimal cost to produce an output that is easily understood by end users. We developed an interpreter that reads the counter-example, using the route table and track layout to generate an interpretation. Only the relevant data is presented to the user in natural language and in a tabular form. By exploiting domain knowledge the interpreter significantly simplifies the output from NuSMV providing a general description of the error. In some cases the output is more specific and tells the user exactly what the error is in the specification.

This paper discusses why animation was not suitable in our application and describes the details of our interpreter. By providing a domain-specific interpretation, we show how we can present counter-examples to the user in the natural language of their domain. We suggest for applications in which end users have little knowledge of model checking, that a domain-specific interpretation provides superior feedback. The results can be tailored to the domain and resented using language consistent with domain terminology. Section 2 describes the different train models and safety properties generated by the verification manager, and how these affect the readabil-

ity of counter-examples for domain experts. Section 3 presents our investigation of animation and our prototype for interpreting counter-examples. Section 4 describes our implementation for interpreting counter-examples, and Section 5 describes the evaluation of the tool's effectiveness. We discuss related work in Section 6 and summarise the results in Section 7.

2 Modelling the Railway Interlocking

The model and property being verified have significant impacts on the readability of counter-examples. The further a model is abstracted the more difficult it becomes to interpret. Abstraction often involves leaving some data out of the state space of the model. This makes it difficult for experts in the application domain to understand the counter-example when there is information missing that they would like to see. It is also difficult to interpret a counter-example when there is a lot of data that is not relevant to the error in the system.

2.1 Railway Interlockings

A railway interlocking is a system of points, signals, and tracks designed to permit the safe movement of a train through a railway layout. Figure 1 is an example track layout of a railway interlocking. Points (for example, *511*) are movable components on a track, and can be set reverse allowing a train to change from one track to another at low speed, or set normal allowing the train to continue in its current direction at high speed. Signals (for example, *5*, *6*, *7* and *8*) use colour indications (for example, red for stop) similar to regular traffic lights to indicate to the train driver if they have the authority to proceed or if they have to stop. Track circuits (for example, *1C*, *5A*, *8B*, *8A*, *7A*, *8D* and *8C)* are sections of track in the railway interlocking.

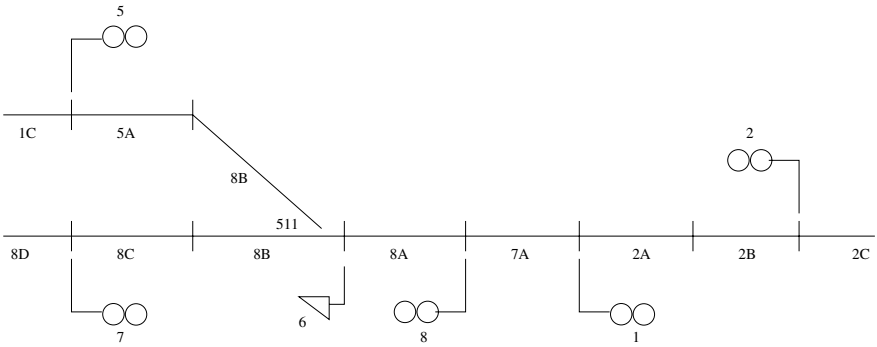

Figure 1. A small track layout.

Table 2 is part of the signal control table corresponding to Figure 1. A route is a path between two facing signals. Table 2 contains only one route, *5(1M)*, which starts at signal *5* and ends at *1*. The first number in the route identifier (for example, *5*) is the signal from which the route starts. The second number is unique for every route attached to a signal. For this paper it can be assumed that

it begins at *1* and will be incremented for each additional route. The letter *M* or *S* means main or shunt route respectively. The only shunt routes included in the table are preset shunts. These are shunt routes that behave as a main route. A signal control table specifies the conditions required for locking a route ready for use (set reverse), and for holding a route (holding it reverse). The columns titled "Points Normal", "Points Reverse", "Routes Normal", and "Routes Reverse" include the requirements of the points and routes for locking and holding a route reverse, while the "Maint. By Tracks Occ." column includes the tracks that a train using the route must clear before the state of the corresponding point or route can change. The "Tracks Clear" column provide the tracks that are required clear before a route can be locked reverse.

Signal Route No.	Route To	Requires					
		Points		Routes		Route Holding Maint. By Tracks Occ.	Tracks Clear
		Normal	Reverse	Normal	Reverse		
5(1M)	1		511			5A	5A 8B 8A 7A
				6(2S)		5A 8B	
				8(1M) 8(2M)		5A 8B 8A	
				2(1M)		5A 8B 8A 7A	

Table 2
A small signal control table.

2.2 The Train Models

The verification manager generates two different models described in the NuSMV input language. The different models are used to detect different errors in the signal control tables. All models include a model of the interlocking system and either one or two trains. System requirements, or safety properties, are modelled as simple invariants [14], for example, a train must not derail. Models are generated for a specific *verification area,* which is a section of a railway interlocking. At the very least a verification area should include one route and all of its opposing routes. For the purposes of modelling we divide track circuits into segments, each segment defining a unique traversal of the track circuit. A segment, as shown in Figure 2, is depicted by the dashed arrow lines such as *8Ba*, *8Bb*, *8Bc*, and *8Bd*. The models generated by the verification manager include train(s) and their positions (the track segment they are occupying), points and their lie (normal/reverse), and routes and their locking (normal/reverse). There is a variable, *currentRoute*, for each train in the model that represents the route that the train is currently using. The rules for changing the state of the variables come from the *Signalling Principles* [11] and are

dependent on the current value of the state variables. The rules for changing signal aspects in our abstracted model only depend on the values of other state variables and not the current aspect. We therefore exclude signals from the state space for efficiency, and they are modelled by checking the current values of the variables that the signal depends on. Trains are well-behaved in the model. They travel along an interlocking obeying the rules in the Signalling Principles. They stop at red signals and can also stop on any track segment. There is no speed associated with train movement.

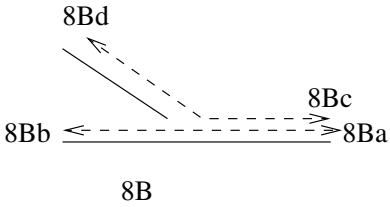

Figure 2. An example of track segments for track *8B*.

There are two basic models. The first model we call the *TC* (tracks clear) model and includes a complete representation of the control tables, that is, all signalling equipment and locking data from the control tables. This model is used to check for collisions between trains travelling in the same direction. The second model we call the *NTC* (no tracks clear, meaning tracks clear are excluded) model. This model excludes the tracks required clear column in the signal control table in order to eliminate redundancy, and tests route holding conditions. Trains are allowed to occupy the same route and even the same segment at the same time. The model is used to check for derailments and collisions between trains travelling in different directions.

There are four safety properties/invariants that we call *derail*, *opplock*, *diffpath*, and *samepath*. The *derail* invariant uses a one-train *NTC* model and checks for train derailments when a point is set incorrectly or moves underneath a train. The *opplock* invariant is used with a one-train *NTC* model and states that whenever a train is on the first track of an opposing signal (that is, a signal set for the opposite direction), there is not a route set reverse from that signal. The *diffpath* invariant is used with the *NTC* model and two trains. It is used to check for head-on collisions and side swipes. The *samepath* invariant is used with the *TC* model and two trains. It is used to check for trains running into the back of each other. Each of the invariants are used to detect different errors in the control table. There is an overlap between the types of errors that *derail*, *opplock* and *diffpath* will detect, but to detect as many errors as possible each invariant must be verified.

2.3 Effect of the Model Design on Interpreting Counter-Examples

The design of the model directly affects the counter-examples that are generated. For example, early models produced by the verification manager used a variable for every route in the verification area that represented how far through the route the train had travelled. This was replaced by the *currentRoute* variable discussed in

Section 2.2. The current route variable reduced the number of states and irrelevant variables in the counter-example and made the raw data more readable. However, the counter-examples were still difficult for signalling engineers to interpret.

Abstracting the model can make the counter-example more difficult to read. Our model abstracts by excluding signals. This reduces the number of state variables but also means that the user does not know what the aspects of any of the signals are. We have proven through comprehensive testing that the model is accurate. However, signalling engineers still find the model disconcerting.

The *NTC* model is used for the *derail, opplock,* and *diffpath* models. This model excludes tracks clear and is specifically designed to check errors in the route locking and holding conditions. The model permits two trains to lock and use the same route and occupy the same track segment at the same time so that it is able to check other locking conditions in the signal control table. Errors that allow trains to occupy the same track segment are checked by the *TC samepath* model. When signalling engineers see two trains occupying the same track segment they immediately assume that there is an error. When they find that the control table does not allow this behaviour and without a thorough knowledge of how the model works, they will think that it is incorrect and discard the counter-example. It is therefore necessary to try to hide this detail from users who do not understand the model.

Counter-examples typically include in each state the variables that have changed since the previous state. When verifying railway interlockings, to check just one route it is necessary to include all opposing routes. If an error is detected, the counter-example produced will contain variables for every route, track segment, point, train, the current route of each train and so on, even if it is only a single error that occurs in one route. This can be overwhelming given that a small verification area can contain as many as 30 variables.

3 Approaches to Presenting Counter-Examples

There are a number of approaches to presenting counter-examples to the user, each of which can be assessed in terms of the cost to produce the counter-example and the effectiveness of the approach. The most obvious solution is to present the raw data produced by the model checker; the cost is nil but the effectiveness is unsatisfactory for reasons already discussed. Other approaches involve a level of translation to present the data in a more intuitive way. However, when interpreting abstracted models, the more translation required, the greater the risk that an incorrect interpretation will be produced. We investigated two approaches, animation and interpretation, to make counter-examples more informative. Both approaches were intended to be fully automated and integrated with the verification manager and the *Signalling Design Toolset*.

3.1 Animation

Animation is an excellent, user-friendly approach to provide visual feedback to the user. A feasibility study was undertaken to investigate animation of counter-examples and its cost-effectiveness. Animating an abstracted model involves a number of problems. For a railway interlocking it is necessary to animate all signal, route and point lockings. Our models do not include signals. It would therefore be necessary to simulate signal behaviour. The only option for doing this would be to use QR's *Signalling Principles* [11]. This introduces the risk that the signals are modelled incorrectly according to the actual behaviour of the model. The animation in this case may mask the error or simulate unexpected behaviour. Animating counter-examples produced by the *diffpath* invariant is also problematic. The animation would show two trains using the same route and occupying the same segment at the same time. The user may incorrectly think there is an error as soon as this happens, however this is not the case. For these reasons, animation is unsuitable for our application.

3.2 Interpretation

The second approach involves the automatic interpretation of counter-examples to determine the actual error that caused the collision or derailment. The key concept is to design an algorithm to interpret the counter-example and produce a description of the error that caused the safety violation. The algorithm needs to be generic to handle all of our counter-examples despite the differences between them. Only the data relevant to the error should be gathered and presented to the user. To be of value, the output of the algorithm should provide as much detail as possible without misleading the user.

We investigated the approach to see if we could reliably identify the error that caused a counter-example. We designed a prototype algorithm, or interpreter. By using domain knowledge we were able to locate the error in terms of routes and the lie of signalling equipment. In some counter-examples we could determine the exact error, and for the others we produce a more general result in terms of the lockings of signalling equipment. After trials with Queensland Rail we determined that the results of the interpretation are best presented to the user in tabular form. We discuss the details of the implementation in Section 4.

4 Interpreting Counter-Examples

The concepts described in Section 3.2 were prototyped for proof of concept. The interpreter is a single algorithm used for each of the invariants described in Section 2.2. The interpreter requires three inputs:

(i) The counter-example as a text file.

(ii) The control table specification as an XML file.

(iii) The counter-example type being interpreted.

The *samepath* invariant can be checked at any time. The *derail*, *opplock*, and *diffpath* invariants must be run in this order so that the types of errors detected by each invariant are limited and some errors can be ruled out. For example, when checking the *diffpath* invariant there should be no incorrect point settings. Control tables are also required to be reasonably well-formed. The *Signalling Design Toolset* produces acceptable control tables as it performs a number of checks before the model checker is run [13].

The interpreter performs three basic steps. First the state in which the most recently used route was locked is found and the data relevant to the error is retrieved from this state. The data gathered is then interpreted and in the last step is formatted and presented to the user in a table. A generic format for the output was designed. An example is shown in Table 3 and corresponds to the layout in Figure 1. Included is a set of typical entries for illustrative purposes. The results are presented in the way that signalling engineers currently test railway interlockings. The first row is the route being tested for errors. It is expected that the error will be found in this route. However, it is possible for the error to be in an opposing route. This only applies to the *opplock* and two-train counter-examples as there are no opposing routes in the *derail* counter-examples. The second row includes the location of trains, locking of routes, and lie of points relevant to the error that was found, and the corresponding action that permits the error to occur, such as a request to set an opposing route. The result of the action is presented in the third row; it is a direct consequence of the action and may include a route being set or points changing lie. The fourth row describes the type of error that occurred. The default is to describe the error in a general manner to avoid incorrect conclusions, but in some cases the specific error in the table can be identified.

Route Tested	5(1M)
Action	5(1M) used with 8A track occupied. Request 8(1M) reverse.
What Happened	8(1M) set reverse.
Error	8(1M) set reverse while 5(1M) in use.

Table 3
Template of the interpreted output of a counter-example.

Table 3 is an example interpretation of a collision detected on track *8A*. The cause of the collision is *8A* missing from the "Maint. By Tracks Occ." column of the signal control table in Table 2. The interpreted output of the counter-example states that route *5(1M)* from signal *5* is currently in use, a train is occupying track *8A*, and route *8(1M)* is permitted to be set reverse. Domain experts can use this information to determine where the error has occurred and quickly locate and correct the problem. For the table to be useful, the error must be either in route *5(1M)* or *8(1M)*, otherwise the interpretation would be of less use than the counter-example produced by NuSMV.

The data required for interpretation varies for each of the invariants. A description of the data required and the semantics follows.

derail

- *point* - the point that the train derailed on.
- *pointMoved* - a boolean that is *true* if the point moved underneath the train or while the train was using *route* or *false* if the point was already set incorrectly.
- *trackOccupied* - the track occupied when the point moved underneath the train or while the train was using *route*. This is only used when *pointMoved* is *true*.
- *route* - the route that the train was using when it derailed.

opplock

- *route* - the route that the train was using when the opposing route was set.
- *trackOcc* - the track occupied by the train when the opposing route was set.
- *oppRoute* - the opposing route that was set reverse.

diffpath* and *samepath

- *primaryRoute* - the route a train has been using as its current route for the most number of state transitions when the invariant is violated.
- *secondaryRoute* - the route the other train has been using when the invariant is violated.
- *trackOccupied* - the track occupied by the train on the *primaryRoute* immediately before the state in which the secondary route was set reverse.

The data is interpreted separately for each type of counter-example. The counter-examples may have more than one format for the output depending on the value of some of the data. The format and an example of the interpretations for each type of counter-example are given below.

4.1 Derail Interpretation

There are two behaviours that distinguish *derail* counter-examples. Either the point is set incorrectly before the train crosses it, or the point moves while the train is crossing it. If the value of *pointMoved* is false then we know that the point lie was incorrect when the route was set. However, if *pointMoved* is true we know that the point moved while the train was in-route. We use a separate format in each case to express the counter-example. Table 4 is the format used to present the counter-example to the user when *pointMoved* is false and Table 5 is the format used when *pointMoved* is true. The variable X is used to indicate the lie, or change of lie, of a point. Its value can be normal or reverse. Tables 6 and 7 are examples of the two formats given in Tables 4 and 5 respectively. The counter-examples produced are based on the signal control table from Table 2 and the track layout from Figure 1.

Route Tested	*route*
Action	Request *route* with *point* points set *X*.
What Happened	*route* set reverse.
Error	*point* points not required *X* for setting *route*.

Table 4
Template of the interpreted output in which a point is set incorrectly.

Route Tested	*route*
Action	*route* used with *trackOccupied* track occupied. Request *point X*.
What Happened	*point* set *X*.
Error	*point* points not held by *route* while track *trackOccupied* occupied.

Table 5
Template of the interpreted output in which a point moves while the train is in-route.

Route Tested	5(1M)
Action	Request 5(1M) with 511 points set normal.
What Happened	5(1M) set reverse.
Error	511 points not required reverse for setting 5(1M).

Table 6
Example of the output produced when a point is set incorrectly.

Route Tested	5(1M)
Action	5(1M) used with 5A track occupied. Request 511 normal.
What Happened	511 set normal.
Error	511 points not held by 5(1M) while track 5A occupied.

Table 7
Example of the output produced when a point moves while the train is in-route.

4.2 Opplock Interpretation

Opplock counter-examples have one interpretation. The invariant detects three types of errors, but the output from the interpreter is generic so that it allows the

user to easily determine the problem. A template and accompanying example are provided in Tables 8 and 9 respectively.

Route Tested	*route*
Action	*route* set reverse with *trackOccupied* track occupied. Request *oppRoute* reverse.
What Happened	*oppRoute* set reverse.
Error	*route* used with *trackOccupied* occupied. *oppRoute* set reverse.

Table 8
Template of the interpreted output for the *opplock* counter-example.

Route Tested	5(1M)
Action	5(1M) used with 8B track occupied. Request 6(1S) reverse.
What Happened	6(1S) set reverse.
Error	5(1M) used with 8B occupied. 6(1S) set reverse.

Table 9
Example of the interpreted output for the *opplock* counter-example.

4.3 Diffpath Interpretation

Diffpath counter-examples have one interpretation. The invariant detects two types of errors, but the output of the interpreter is generic to allow the user to easily determine the problem. A template and accompanying example are provided in Tables 10 and 11 respectively.

Route Tested	*primaryRoute.*
Action	*primaryRoute* set reverse with *trackOccupied* track occupied. Request *secondaryRoute* reverse.
What Happened	*secondaryRoute* set reverse.
Error	*primaryRoute* used with *trackOccupied* track occupied. *secondaryRoute* set reverse.

Table 10
Template of the interpreted output for the *diffpath* counter-example.

Route Tested	5(1M).
Action	5(1M) set reverse with 8B track occupied. Request 2(1M) reverse.
What Happened	2(1M) set reverse.
Error	2(1M) used with NG8B track occupied. 2(1M) set reverse.

Table 11
Example of the interpreted output for the *diffpath* counter-example.

4.4 Samepath Interpretation

Samepath counter-examples have one interpretation. In every case the exact error can be detected by the interpreter. A template and accompanying example are provided in Tables 12 and 13 respectively. The error stated on the last line informs the user about the track that is missing from the tracks clear column of the signal control table.

Route Tested	*route.*
Action	*trackOccupied* occupied. Request *route* reverse.
What Happened	*route* set reverse.
Error	*trackOccupied* not required clear for setting *route.*

Table 12
Template of the interpreted output for the *samepath* counter-example.

Route Tested	5(1M).
Action	1C occupied. Request 5(1M) reverse.
What Happened	5(1M) set reverse.
Error	1C not required clear for setting 5(1M).

Table 13
Example of the interpreted output for the *samepath* counter-example.

5 Evaluation

To evaluate the interpreter a total of 177 counter-examples were generated and interpreted using our prototype. The control tables of a small verification area was used. Counter-examples were generated by injecting errors into the control table. A single element was removed such as a track required clear, then the model was built using our prototype model builder and run using NuSMV. This was done iteratively until each element in the control table was removed exactly once. Exactly one error was injected for each test. The interpreted results were manually examined against the error that had been injected.

Table 14 summarises the results for each of the invariants. Of the 177 counter-examples, the interpreter correctly identified the route containing the error in 174

of the cases. The remaining three counter-examples were produced by the *opplock* invariant and the error was found in the the opposing route (*oppRoute*). In each cases the user is still able to determine what the error was and the interpretation provided better feedback than the raw data. Six out of 13 of the interpretations for the *derail* counter-examples determined the exact error in the control table. The remaining seven interpretations were more generic and the error was found in the primary route. The exact error was determined for all of the errors produced by the *samepath* invariant.

Model	Tests Run	Error in Primary Route	Error in Secondary Route	Identified Exact Error
derail	13	13	0	6
opplock	55	52	3	0
diffpath	58	58	0	0
samepath	51	51	0	51
Total	177	174	3	57

Table 14
Results of evaluation

Users at Queensland Rail were shown the raw data output from NuSMV, and the interpretations produced by our interpreter. Early in the development of the verification manager, domain experts at Queensland Rail struggled to understand the raw data produced by NuSMV. It took users a considerable amount of time to understand the train movements and locate the route and interlocking equipment relevant to the error. The problem was further complicated when the user realised that different trains are allowed to occupy the same track at the same time. Signalling engineers who had never seen the counter-examples were used to test the interpretations. Initially they were resistant to having to use interpreted data and questioned why they could not just see the raw data. They felt they had the knowledge to be able to understand the raw data. However, after looking at our interpretations they could quickly tell us where they would look for the errors. The interpretations improved the speed in which the user could locate the error. There was the added benefit that the user did not have to see that two trains are allowed to travel on the same track at the same time and that there are no signals in the model.

6 Related Work

Counter-examples are known to be a usability problem not just for domain experts but also for engineers designing the models. Most research, for example Clarke et al. [3], Pasareanu et al. [10] and Dwyer et al. [5], focuses on eliminating false or spurious counter-examples rather than improving their readability. These papers all

address false counter-examples produced due to abstraction. This is not a concern for us as our abstractions are precise leaving the behaviour of the model largely unchanged.

Huber [7], who also modelled railway interlockings, acknowledged that counter-examples are a problem for the usability of model checking. He suggests that output from model checkers can be tailored to the application area and that only modest efforts are required to modify the model checker and greatly improve the usability. While the paper indicates Huber did not test his suggestions, the approach is similar to our work except we modify the output rather than the model checker itself.

Loer and Harrison [9] discuss the usability issues of model checking. Their approach is to provide a toolkit that is based around the SMV model checker. Their research on interpreting counter-examples was in its preliminary stages, but one of their aims was to support visualisation of error traces. Their approach is general in an attempt to support all problems industrial designers might have, while we focus on a domain-specific application to customise the output specifically for signalling engineers. The paper describes a number of prototyped approaches including tables, natural language, scenario templates, scenario scripts, message sequence charts, operation sequence diagrams and animation. They stated that participants found that tables and operation sequence diagrams were the most useful. The output we produce is a combination of tables and natural language scenarios. Operation sequence diagrams are more complex than needed in our application, as we only need to show the result of a request placed on a point or opposing route.

Ball et al. [1], Groce and Visser [6], and Jin et al. [8] provide general approaches to improving counter-example readability. Their work is interesting because it is not domain-specific, however, the final counter-example is still presented in a form that relies on the user understanding counter-example traces, and to some degree the model. Ball et al. describe an approach in which multiple traces describing different errors are produced. Their algorithm exploits the existence of correct traces to identify transitions that are not in the error trace. This helps to localise the error and improve the output to the user. Groce and Visser use a similar approach of searching for negative and positive sets of executions. A negative execution is one in which the same error is produced with a different trace. A positive execution is a variation on a negative one in which the error does not occur. By comparing the sets they show that common features and differences can be used to provide more useful feedback than just presenting the original counter-example. Jin et al. describe an approach in which the counter-example is broken into *fated* and *free* segments. Fated segments are actions in a trace towards an error that can not be avoided while free segments are choices that may prevent the error if they are avoided. They annotate a counter-example to describe the unavoidable segments and leave it to the user to decide if the free segments are the fault in the system of if the fated actions could have been avoided.

The general approach used in each of [1], [6], and [8] is to improve the counter-example/feedback in order to provide good counter-examples that include more information related to the error, and less irrelevant information. In our case a

single counter-example provides the necessary information to locate the error. From a single trace we can determine which are the relevant variables and then tune the output specifically for our targeted users without the overhead required in the above approaches. While simplifying the counter-examples would aid our users, it is the behaviour and lack of understanding of the model that is our main concern. Our end users are also not programmers or Software Engineers and they are not necessarily familiar with the output from model checkers or compilers.

7 Discussion

Two approaches, animation and interpretation, were considered for presenting counter-examples to end users. Animation was investigated but is not appropriate for our application as our model is significantly abstracted from the real system. An interpreter was designed and prototyped based on the *Signalling Principles* [11]. This creates a robust algorithm that can handle any counter-example from our models. The interpreter searches for key data in the counter-example and presents this in an interpreted form to the user and excludes all the irrelevant details that often confuse the user. It was only by using domain knowledge that we could produce improved interpretations.

To evaluate the interpreter test cases were generated by injecting single errors in the control table. This approach is reasonable since model checkers stop at the first counter-example they find and so it is expected that any counter-examples produced in practice should not vary from the ones we produced. We also expect that the general behaviour of the interlocking will still follow the *Signalling Principles*, and the output of the interpreter should not vary. However, we still did not test the prototype against counter-examples found in practice. This was not possible since the verification manager is in its preliminary stages and no real counter-example data is currently available.

References

[1] Ball, T., M. Naik and S. K. Rajamani, *From symptom to cause: localizing errors in counterexample traces*, in: *Proc. of ACM Symp. on Principles of Programming Languages*, New York, NY, USA, 2003, pp. 97–105.

[2] Cimatti, A., E. Clarke, F. Giunchiglia and M. Roveri, *NuSMV: A new symbolic model verifier*, in: *Proc. of Int. Conf. on Computer Aided Verfication, CAV'99*, LNCS **1633** (1999), pp. 495–499.

[3] Clarke, E. M., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Counterexample-guided abstraction refinement*, in: *Computer Aided Verification*, 2000, pp. 154–169.

[4] Clarke, E. M., O. Grumberg and D. A. Peled, "Model Checking," MIT Press, 1996.

[5] Dwyer, M. B., J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, H. Zheng and W. Visser, *Tool-supported program abstraction for finite-state verification*, in: *ICSE '01: Proc. of the 23rd Int. Conf. on Software Engineering* (2001), pp. 177–187.

[6] Groce, A. and W. Visser, *What went wrong: Explaining counterexamples*, in: *Proc. of the 10th Int. SPIN Workshop on Model Checking of Software*, Portland, Oregon, 2003, pp. 121–135.

[7] Huber, M., "Towards an Industrially Applicable Model Checker for Railway Signalling Data," Master's thesis, University of York (2001).

[8] Jin, H., K. Ravi and F. Somenzi, *Fate and free will in error traces*, in: *Proc. of the 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (2002), pp. 445–458.

[9] Loer, K. and M. Harrison, *Towards usable and relevant model checking techniques for the analysis of dependable interactive systems*, in: *Proc. of 17th IEEE Int. Conf. on Automated Software Engineering (ASE 2002)* (2002), pp. 223–226.

[10] Pasareanu, C. S., M. B. Dwyer and W. Visser, *Finding feasible counter-examples when model checking abstracted Java programs*, Lecture Notes in Computer Science **2031** (2001), pp. 284–298.

[11] Queensland Rail Signal and Operational Systems, *Signalling Principles - Brisbane Suburban Area*, Technical Report S0414, Queensland Rail Technical Services Group (1998).

[12] Robinson, N., D. Barney, P. Kearney, G. Nikandros and D. Tombs, *Automatic generation and verification of design specification*, in: *Proc. of Int. Symp. of the International Council On Systems Engineering (INCOSE)*, 2001.

[13] Tombs, D., N. J. Robinson and G. Nikandros, *Signalling control table generation and verification*, in: *Proc. of Conf. on Railway Engineering (CORE 2000)* (2002).

[14] Winter, K., W. Johnston, P. Robinson, P. Strooper and L. van den Berg, *Tool support for checking railway interlocking designs*, in: *Proc. of the 10th Australian Workshop on Safety Related Programmable Systems*, Sydney, Australia, 2005.