Verifying Relay Circuits using State Machines

P.H.J. van Eijk
Utrecht University, Department of Philosophy
Heidelberglaan 8, Utrecht, the Netherlands
http://www.phil.ruu.nl/home/pve
and
EDS, PO Box 2233, Utrecht, The Netherlands
pve@cvi.ns.nl
February 1997

Abstract

In this paper we present, illustrate and discuss a number of techniques that can be used in the modelling and verification of electro-mechanical relay circuits. These techniques are based on state machine descriptions of circuits and their functions, and on applying validation tools for properties of such descriptions. In particular we have applied tools that are based on the PROMELA language.
Key words & Phrases: formal specification, formal verification, safety critical system, relay, state machines, PROMELA.

1 Introduction

For any artefact that is designed, the question sooner or later comes up: 'If we build it this way, will it do what we want it to do?' This is the correctness question, and the answer to it can be a matter of life and death. In this paper we will see how current technology for verifying correctness can be applied to relatively old hardware technology. It will become apparent that correctness of relay circuits is a topic with many sides, and many levels. On each of these levels a number of techniques can be applied.

This paper is subdivided in the following sections:

- An overview of electromechanical relay circuits.
- An example: train mode control, and its combinatorial correctness.
- PROMELA [4]
- The example studied as a sequential system.
- Verification against state machine models.
- Conclusion.

Surprisingly, little literature exists on the verification of relay circuits. It is discussed in the context of safety interlocks by Jacky in [3]. That work concentrates on *combinatorial* correctness, where the past history of inputs has no influence on the output of the system and only the current input is relevant.

In this paper we also discuss the correctness of *sequential* systems, of which the behaviour depends on the history of inputs or on the interleaving of events. Such systems exhibit state. To our knowledge, there is little further literature concerning formal approaches to relay circuits.

2 Electromechanical relays

An electromechanical relay, or relay for short, is a hardware device that can implement a number of digital switching functions. It typically consists of an electric coil that can exert a magnetic force on a number of contacts. When electrical power is applied on the coil the contacts either break or make. In short, a relay is an electrically operated switch. Logically, the relay can implement a decoupler or negation function, and the wiring can implement the logical 'and' and 'or' functions. In this way circuits can be built that implement arbitrarily complex propositional logic formulae.

Relays have been in use for over a hundred years now. Entire telephone switches were once built out of them. Despite their relative demise as the result of electronics, and programmable electronics in particular, there are still applications where their ruggedness and simplicity is vital. For a number of these applications the same reasons that lead to the deployment of relays make that the correctness of the circuit is safety-critical.

A typical design using relays consists of a network of interconnected switches (contacts), including the inputs of the system, that terminates in the relays. A relay circuit is usually described with a graphical notation that illustrates the wiring between the components, but not necessarily the physical layout.

There is an almost infinite variety of relay types. The number of contacts can differ, as well as the order of their opening and closing. Time-relays exist, that delay their closing for anything from seconds to hours, and memory-relays, with two coils, serving as a latch.

In figure 1, a simple relay circuit is drawn. D is the coil of the relay, and A, B and C are contacts or switches. In this example there is no causal relation between A, B, C, and D other than the one shown in the diagram. The notational convention in these diagrams is that the contacts that belong to a relay are labelled with the same letter, so for example a relay labelled A governs all contacts that are labelled A. The top and bottom lines are wires that power the circuit. A is a *normally-closed* contact, which on activation will interrupt the current. B and C are *normally-open* contacts, which on activation will allow current to flow. The drawing convention is to show the nonactivated state. In this paper the two cases are also distinguished by the fact that normally-open contacts are shown to the left, and with a wider gap.

Logically this circuit can be described as

```
D = !A && (B || C)
```

where `!` denotes logical `not`, `&&` denotes logical `and`, and `||` denotes logical `or` (this is

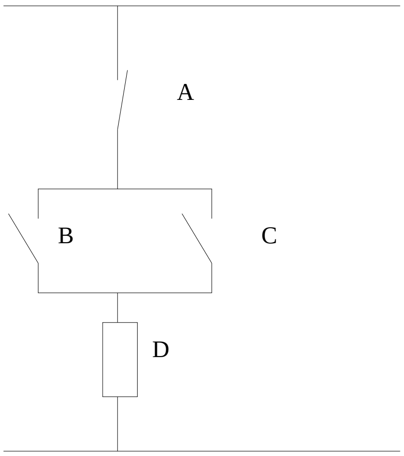

Figure 1: Example relay circuit.

PROMELA notation, to be explained more fully later). In words, the relay D is activated when A is closed and either B or C is closed.

3 An example

Like an automobile, a railway train has several modes of operation. A car can be off, it can have the accessories on, and it can be fully operational. These modes are typically controlled through a single key operated switch. For a railway train the operational modes determine, for example, whether or not the heating and lighting systems are on, how the doors can be operated, and whether the train can actually move. A typical train can have as much as six major modes. The system that realises the control over these modes is called the *operational mode control system* (in Dutch: bedrijfstoestandskeuze systeem).

The passenger train in this example can consist of a fairly arbitrary number of motorcars (each car has its own engine, like a subway, as opposed to the case of a train that is pulled by a separate locomotive engine), each of which has two operator cabins, one on each side. The entire train can be controlled from each cabin, subject to certain requirements. Some of these requirements are safety-critical. For example, the train must only be able to move if it is in the 'in service' mode, which can only be entered when a key operated switch is activated. The first operator to activate his key must gain control, and a second operator (in a different cabin) must be denied control. Computer scientists will recognise this as a mutual exclusion problem.

The operational mode control system is typically implemented using relays, although in newer designs PLC's (Programmable Logic Controller, a type of microcomputer) are increasingly applied. All cabins are equipped with identical copies of a relay circuit. These copies are connected by a set of wires that run through the entire length of the train. Together these copies perform the functions of the operational mode control system. The requirements can be divided in those that apply to each individual copy of the circuit and those that apply to the system as a whole. For example, there may be some kind of exclusion within each individual copy required, as well as exclusion between all copies.

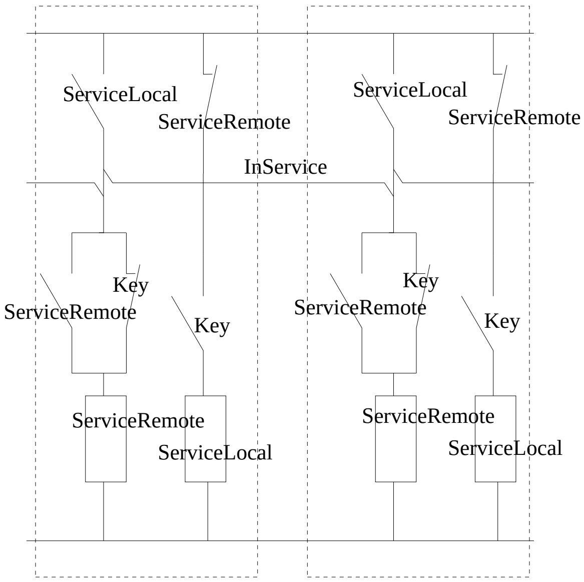

Figure 2: Two copies of a control circuit

One of the complications is that the system can consist of a (relatively) arbitrary, though finite, number of connected copies of the circuit. One would want to validate the system as a whole, independent of the number of connected copies.

3.1 The example as a relay circuit

The circuit that we take as an example here, is a simple operational mode control system. It controls the 'service' state of a train. In each cabin a train operator can attempt to set the train in the *in service* state through the use of a key-operated switch. The function of the circuit is to allow only *one* operator to actually control the train, i.e. the circuit implements a mutual exclusion. The circuit is drawn in figure 2.

The system consists of a number of identical copies of a circuit. Here two copies are drawn. Each cabin is equipped with one such circuit. The top and bottom horizontal lines are the electrical power feed, and the middle horizontal line (labelled 'InService') connects all cabins together. This line signals whether or not the entire train is in service, it also functions as a n-ary OR that is distributed over the entire train. Presumably, other train systems, such as the door and traction systems, are controlled on the basis of this signal.

The relation between the contacts and the relays governing them is indicated by the name that they are labelled with. We will append a digit to the names to indicate the copy of the circuit that they are in, e.g. `servicelocal_1`. All contacts are shown in the state in which the relays are not activated.

So, for example, if `Key_1` is activated, both its contacts will move. This then activates the relay `ServiceLocal_1` and its only contact. Through the wire `InService` this is signalled to the entire train, and in all other cabins `ServiceRemote` will be activated. The result of this is that in all those cabins operation of `Key` is ineffective, and an exclusion will have been realised.

3.2 Combinatorial correctness

The combinatorial behaviour of the circuit can be formalised with the following equations:

```
servicelocal_1 = key_1 && !serviceremote_1
serviceremote_1 = inservice && (!key_1 || serviceremote_1)

servicelocal_2 = key_2 && !serviceremote_2
serviceremote_2 = inservice && (!key_2 || serviceremote_2)

inservice = servicelocal_1 || servicelocal_2
```

For any input, all combinations of values that fulfil these equations represent a stable state of the circuit. Conversely, if the equations are not satisfiable for all inputs the circuit is not stable. An example of a system that is not stable is simply:

```
k = !k
```

A relay implementation of this will most likely exhibit an oscillation.

The correctness criteria on the circuit are:

```
!(servicelocal_1 && servicelocal_2)
```

meaning that only one of the cabins can be the operating cabin, and

```
!(servicelocal_1 && serviceremote_1) &&
!(servicelocal_2 && serviceremote_2)
```

which is a type of consistency check on the settings of the relays of one cabin. The relays should not simultaneously indicate that this is the operating cabin, and that some other cabin is the operating cabin.

The behaviour equations, taken as a logical proposition, should logically imply these correctness criteria. It turns our that the first one is not true: there is a race condition if two keys are switched simultaneously.

4 PROMELA

PROMELA[4] is a language designed for the modelling of communicating finite state machines. It was designed for the modelling of communication protocols, but it can be used for the many other parallel or distributed computer systems. Beyond these applications, it can even be used to model systems (such as relay circuits) that are not traditionally seen as computing devices. In this paper we show some modelling levels at which PROMELA can be used to analyse relay circuits.

PROMELA as a language is complemented by SPIN, a software package for the analysis of specifications written in PROMELA. SPIN can be used in a number of ways. In an interactive mode it can provide a simulation of the behaviour of a system. In an automatic mode it can try to exhaustively analyse the statespace of a system, proving or refuting user specified correctness requirements.

A PROMELA model describes a set of process types. Starting a process can happen anywhere in the model. Initially, only the `init` process is active. With the construct

```
ch!grant
```

a process sends the message `grant` to the channel `ch`. A message can also be composed of constants and variables, for example:

```
t2r!report(x,1)
```

With the construct

```
t2r?x
```

a process can receive a message in the variable `x`. After the questionmark a composite message is also possible. A receive command with such a composite message can only be possible if it matches the message in the channel. This can be used in a choice construct such as:

```
if
:: mychan?t(grant) -> i=i+1
:: mychan?t(deny) -> skip
fi
```

Depending on the content of the message, one of the branches of the if-construct will be chosen and the command after the arrow will be executed. The `skip` command is simple: it does nothing. A loop looks like fllows.

```
do
:: r2e?reserve -> ...
:: r2e?release -> break
do
```

Such a loop repeats until a break is performed. The `break` command indicates that the enclosing loop is to be terminated. The `if` as well as the `do` constructs have the property that, if none of the branches is possible (executable), the process in which they are contained blocks until at least one of the branches is executable. (In the meantime another process may be able to progress.) If there are multiple executable branches the model only states that one of them will happen. In a simulation of a model this will be an arbitrary choice. In a validation, all of them will be tried. SPIN does this with a depth-first treewalk.

PROMELA also has variables, expressions, and assignment statements. These are illustrated below. An important property of PROMELA is that sending, receiving, and testing of conditions are also statements. These will block the progress of a process if they are not executable. E.g. in the fragment

```
(3==2); i=0
```

the assignment will never be executed. Note that the `->` operator is semantically equivalent to the semicolon, both indicate a sequence of statements. A sequence of statements can be enclosed in an `atomic` construct, which makes them indivisible, i.e. no statements in other active processes will intervene.

The state of a PROMELA model is made up of the states of each of the active processes plus the values of the variables, including the contents of channels. Each statement (e.g. message, test or assignment) descibes the transition of one state to another. A PROMELA model therefore describes a number of possible ways in which a system can make state transitions and exchange messages. In the following example the value of the variable `foo` in the final state can be one of three (0, 1 and 2) because between the test and the assignment the other process could also perform the test.

```
int foo=1
proctype A() {(foo==1) -> foo=foo+1}
proctype B() {(foo==1) -> foo=foo-1}

init {run A(); run B()}
```

The system as a whole has 9 different possible states, of which 3 are final states. In a simulation we would see one of these final states being reached. A validator looks at *all* possible states, and checks these against correctness criteria given in the model. An example of such a correctness criterion could be described as

```
assert(foo!=1)
```

which would not be the case here. With a validator tool we can therefore try to answer the question whether or not there is a way to reach a certain state.

To summarise the difference between simulation and validation as we use those concepts here: a PROMELA model describes a tree of possible behaviour, where each node in the tree is a state, and each branch a step (event, transition, execution of an atomic sequence). A simulation exercises one path in this tree, a validation attempts to exercise *all* paths in the tree. The validator tool that we have used (SPIN) can, on our Unix workstations (vintage 1992), analyse thousands of states per second.

5 Modelling Sequential behaviour

In the above we have looked at the modelling of relay circuits by simple combinatorial equations. At that level there are a number of correctness criteria that can be expressed, and verified or refuted.

In this view relays are in one of two states: activated or not. Yet, the story is not that simple. All switching elements have delays, and these delays can influence the correctness of the system. What happens in fact is that each switching activity progresses through a sequence of at least three phases or states. Suppose the relay is unpowered, or off. That phase ends when power is applied, and here an intermediate phase starts. From this point on the magnetic force starts to build up, and above a certain threshold the contacts start to move from their resting positions. A 'break' contact will break as soon as it has started to move, a 'make' contact will make as soon as it has reached its end position (we are assuming for the moment that it will not bounce back from there). When all contacts have done so, the intermediate phase is over, and the last phase begins: the relay is truly activated. There is not necessarily a fixed order in which the contacts make or break, although some types of relays have been designed to exhibit an explicit order, e.g. make before break or vice versa.

The fact that relays have this type of behaviour can have an impact on the correctness of the circuit. There can be situations in which there is literally a race between two contacts to reach their positions, and the behaviour of the system depends on the winner of the race. In cases like these, it is thus important in an analysis to consider all possible sequences of operations.

For some types of relays this behaviour is also influenced by ageing. The implication of that is that such systems can fail after a long time of correct operation.

5.1 Modelling Approach

How to model this intermediate stage? There is a parallel between the delay in the closing of a relay contact, and the delay a message can experience on its way from sender to receiver. It is tempting to use this analogy in modelling a relay circuit in PROMELA. A relay is then a process with an incoming message (the electrical current) and a number of outbound messages, one for each contact. The wires would then be the message conduits or channels. Appealing in this approach is that it is truly event based, and that all possible interleavings of events will be considered by the analyser. Unfortunately this approach has its flaws. It is awkward to model junctions of wires (bifurcations) in this way. They will become additional processes with multiple inputs and/or outputs. Some wires are bi-directional (such as `inservice` in the example), which also adds to the complexity.

Instead of looking at events, we can also look at states. The voltage on a wire is a state, represented by a variable, and (re)computing the position of the relay contacts is then a state change, which can be represented by an assignment. Each relay now corresponds to one assignment statement, with the left hand side variable representing the coil, and the right hand side expression representing the combinatorial logic of wires and contacts. Intermediate wires (in the example `inservice`) can also be represented explicitly with an assignment statement.

The behaviour of the system can now be described by a set of assignment statements that are randomly executed. All reachable states are covered by this model.

The example system can be described in PROMELA as follows. All variables in the model are global, in order to be able to inspect and modify them, where appropriate.

```
proctype modecontrol()
{ /* this is the model of the system */
do
::        servicelocal_1 = key_1 && !serviceremote_1; skip
::        serviceremote_1 = inservice && (!key_1 || serviceremote_1); skip

::        servicelocal_2 = key_2 && !serviceremote_2; skip
::        serviceremote_2 = inservice && (!key_2 || serviceremote_2); skip
```

```
::        inservice = servicelocal_1 || servicelocal_2; skip
od
}

proctype monkey()
{ /* this is the model of the environment of the system */
do
::        key_1=1; skip
::        key_1=0; skip

::        key_2=1; skip
::        key_2=0; skip

od
}

init
{       run modecontrol();
        run monkey();
}
```

The `monkey` process[1] can arbitrarily turn all key switches on and off, in any order. Two parallel processes, one representing the system, and the other representing the environment using the system, together describe a state space. (The occurrence of `skip` instructions is for technical reasons. Without them, the process `modecontrol` would have only one control state, which would make the output of SPIN less clear.) With such a model, the behaviour of the system is described on an electromechanical level. Hence SPIN will now be able to explore all behaviour that is electromechanically possible, rather than exploring logically possible behaviour. For example, make before break or vice-versa can be studied, as well as relay deactivation before all contacts are made.

5.2 Sequential Correctness

The correctness requirements on the model are the same as before, and can be described in PROMELA as follows. The `checks` process runs concurrent with the other processes and can intervene at any time to check on the global variables. Although this process can exercise its asserts only once in an execution trace, the validator will ensure that all possible intervention points will be tried.

```
proctype checks()
{
   assert(!(servicelocal_1 && servicelocal_2));
```

[1]Named after the saying: *if you give enough monkeys enough typewriters, you will find a Shakespeare sooner or later.*

```
   assert(!(servicelocal_1 && serviceremote_1));
   assert(!(servicelocal_2 && serviceremote_2))
}
```

Not surprisingly, the same race condition still persists in the system. If the two switches are turned on together (simultaneously), there will be no mutual exclusion. For the system to be really correct, this circuit would need to implement a distributed leader election[1] function) Although this circuit is incorrect in particular circumstances, for some applications these circumstances are so rare that they need not be considered. The absence of these circumstances can be rephrased as assumptions.

The assumption under which the circuit will work correctly, is that no external input will be given as long as the circuit is still processing. In other words, the circuit has to be *stable* or *quiescent* before an additional input is given. In the model this means that every assignment that can have an effect must have been done. This is expressed by translating every assignment of the form `x = E` into a condition `x == E`, and taking the conjunction of all these conditions. In the example this boils down to adding the following PROMELA code.

```
proctype modecontrol()
{
do
/* as before */
::      stable=
        (servicelocal_1 == (key_1 && !serviceremote_1)) &&
        (serviceremote_1 == (inservice && (!key_1 || serviceremote_1))) &&
        (servicelocal_2 == (key_2 && !serviceremote_2)) &&
        (serviceremote_2 == (inservice && (!key_2 || serviceremote_2))) &&
        (inservice == (servicelocal_1 || servicelocal_2));  stable=0
od
}
```

Of all the states of this process, there is a subset of states where another process can assume that the system is stable, and those are the states at the last semicolon, of the above code, where the variable `stable` is true. That also explains the presence of the assignment `stable=0`, without which the process could move halfway into processing an input with the variable `stable` still true.

Expressing that the system will only be operated in a stable state can be done with the following monkey process.

```
proctype monkey()
{
do
::        stable; key_1=1; stable=0
::        stable; key_1=0; stable=0

::        stable; key_2=1; stable=0
::        stable; key_2=0; stable=0
```

```
od
}
```

The keys can only be operated when the system is stable, and because of the assignment `stable=0`, they can be operated only one at a time. The correctness requirements in this model (the `checks` process) need to run uninterrupted by the system process, which is done here with an 'atomic' statement. Otherwise, the requirements could be checked in an unstable state.

```
proctype checks()
{ atomic{
   stable;
   assert(!(servicelocal_1 && servicelocal_2));
   assert(!(servicelocal_1 && serviceremote_1))
        }
}
```

Additional embellishments and refinements can be added to this model. This model assumes that all relays are indeed in their nonactivated state when the system is initialised. This is realistic for normal relays, but not so for memory relays or external switches. It is a small matter to add a nondeterministic assignment of values to variables in the initialisation phase. This would then expose all errors that may be the result of an incorrect initialisation.

In case the correctness of a circuit critically depends on the fact that a certain contact makes before another breaks, this can be expressed by putting the corresponding assignments in sequence rather than in independent branches.

6 Relay circuits as implementations of state machines

On a certain level of abstraction the circuit is a state machine, with external stimuli as true events. Some relay circuits are required to actually remember something, rather than just compute some logical function. It can then be convenient to describe the actual state machine, and check it against the relay circuit. On the state machine itself certain validations are also possible: can every state be reached? Our example system can also be described in a style that is more natural in PROMELA. In this model, messages on a channel `a` reflect the interaction with the system, and corresponds to the `key` variables in the previous model.

```
proctype modecontrol2()
{
idle:
        if
        :: a?keyon; goto ready
        fi;
ready:
        if
```

```
        :: a?keyoff; goto idle
        fi
}
```

where the state labelled `ready` corresponds to the wire `inservice` being activated. The monkey process for this system could be

```
proctype monkey()
{
do
:: a!keyon; a!keyoff
od
}
```

This expresses that a switch cannot be turned off without being turned on. This does not hold for push buttons of course, which can be easily modelled too.

By taking them together, the monkey process and the system process can be used to analyse the system for deadlocks and reachability. Problems in this area might indicate an incomplete system description.

We now have two models of the same system, expressed in different styles. The last description describes the system at the *specification* level, so to say, whereas the first description is an *implementation* description. Hence it becomes possible to compare the two.

For this, two components are needed. One is a checkable claim of correspondence between these two systems, the other is a new monkey process that drives the two machines simultaneously.

The correspondence relation, or equivalence relation, can look like this:

```
atomic{
   stable;
   assert((modecontrol2[i]@ready)==inservice)
        }
```

In the assert, the predicate `modecontrol2[i]@ready` references the state of `modecontrol2`, which is only true if that machine is at the state `ready`. As in the previous section care needs to be taken to ensure that the relay system is in a stable state, when the check is executed.

The monkey process describes a subset of all possible user input to the system. For example, it is not very interesting to switch the key so fast that the system cannot react to it. The monkey process can also be restricted in order to narrow down the search for an error.

```
proctype monkey2()
{
do
::      atomic{ stable&&(key_1==0); key_1=1; stable=0; a!keyon}
::      atomic{ stable&&(key_1==1); key_1=0; stable=0; a!keyoff}
od
}
```

This description expresses the mechanics of the inputs and at the same time states what should be considered equivalent events. The inputs need to be atomic in order to force the machines to move before an equivalence check is made.

7 Concluding Remarks and Future Research

Relay systems can be analysed in great detail, right down to the level of 'meaningless' errors. The main idea explored here is that there are a number of ways in which relay circuits can be modelled with finite state machine techniques and tools, and that various correctness checks can be performed on these models. Additionally, two models that have substantially differing styles can be compared automatically for behavioural equivalence. Such styles can correspond to descriptions on the specification and implementation level, respectively.

The main open question seems to be what guidelines should be used in applying these techniques. The precision with which these models can be made leads to the discovery of problems in systems that are so subtle that they are not necessarily considered harmful in practice. An example of this is a race condition between two manual inputs. Also awkward in its treatment is the stability requirement on correctness claims. Perhaps more convenient notation can be invented in order to avoid cluttering up the PROMELA models with atomic statements.

Acknowledgements: Jan Friso Groote, Pim Kars en Bas van Vlijmen commented on earlier versions of this paper. Andre Dirks and Lex Frunt of Dutch Rail introduced me to the system that inspired this paper.

References

[1] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, (3):245–260, 1982.

[2] J.F. Groote, J.W.C. Koorn and S.F.M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. Compass '95 Proceedings of the Tenth Annual Conference on Computer Assurance pp 57–68, IEEE, 1995.

[3] J. Jacky. Verification, Analysis and Synthesis of Safety Interlocks. Technical Report 91-04-01, Radiation Oncology Department, University of Washington, 1991

[4] G. Holzmann. Design and Validation of Computer Protocols. Prentice Hall, New Jersey, 1991, ISBN 0-13-539925-4.
`http://netlib.bell-labs.com/cm/cs/what/spin/index.html`