Int J Software Informatics, Vol.3, No.2-3, June/September 2009, pp. 241–272 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
 Tel: +86-10-62661040

Developing a Domain Model for Relay Circuits*

Anne E. Haxthausen

(Informatics and Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark)

Abstract In this paper we stepwise develop a domain model for relay circuits as used in railway control systems. First we provide an abstract, property-oriented model of networks consisting of components that can be glued together with connectors. This model is strongly inspired by a network model for railways made by Bjørner et. al., however our model is more general: the components can be of any kind and can later be refined to e.g. railway components or circuit components. Then we show how the abstract network model can be refined into an explicit model for relay circuits. The circuit model describes the statics as well as the dynamics of relay circuits, i.e. how a relay circuit can be composed legally from electrical components as well as how the components may change state over time. Finally the circuit model is transformed into an executable model, and we show how a concrete circuit can be defined, checked to be legal, and the reaction to an input can be simulated.

Key words: domain modelling; formal methods; RAISE; relay circuits

Haxthausen AE. Developing a domain model for relay circuits. *Int J Software Informatics*, **2009, 3(2-3): 241–272. http://www.ijsi.org/1673-7288/3/241.htm**

1 Introduction

Motivation Railway interlocking systems are used to control signals and points of railways in such a way that train collisions and derailments of trains are avoided. At many Danish railway stations the interlocking systems are still implemented using electrical circuits containing relays. Such relay systems are documented by diagrams of the electrical circuits, and currently the only way to analyse how they work is to inspect the diagrams and manually draw conclusions. This is hard to do as the number of diagrams for a single system is very high and the logic described in each of them is complicated with many mutual dependencies. Certainly such a manual analysis is not only difficult and time consuming, but may also be error prone. This is not satisfactory for a safety-critical system. Therefore we have started a project, the goal of which is to develop computer-based tools for automating such analyses. The tools should comprise a simulator that can visualise the dynamics of relay circuits, and tool support for formal verification.

Approach For the development of such tools our approach is to follow the TripTych dogma by Dines Bjørner (see for instance Ref.[1]) making a domain model describing the concepts of the application domain prior to the actual development of applications. Apart from separating the concern of describing *what there is* from the concern

of describing *what there should be* (the applications), this ensures that different applications are based on the same conceptual understanding and can reuse basic functions expressed over the domain model.

In this paper we will describe how a domain model for relay circuits can be stepwise developed, and we will show how the model can be instantiated and executed for a concrete relay circuit. For the development we have used the RAISE[2−4] formal method, language and tools. We have chosen to use a *formal* method as formal specifications are unambiguous and allow for formal verification. This is especially relevant for the development safety-critical applications like relay interlocking systems and is recommended by the higher software safety integrity levels of internal standards such as the CENELEC[5] standards for railway applications. RAISE allows for *stepwise refinement*. Refinement is a verifiable transformation of an abstract (high-level) formal specification into a concrete (low-level) executable specification or program. Stepwise refinement allows this process to be done in stages. One of the advantages of using stepwise refinement is that of abstraction: One can start specifying the essential, generic properties of a system without being implementation biased. Design decisions (such as choice of algorithms, data structures and concrete data) can be deferred to later refinement steps, as we shall see in this paper. Stepwise refinement also supports re-use: A common abstraction can be refined in different ways for different applications so that one does not need to start from scratch for each application, as we shall also see in this paper. Each refinement step must be verified to be correct which means that properties of the more abstract specification are preserved by the more concrete specification.

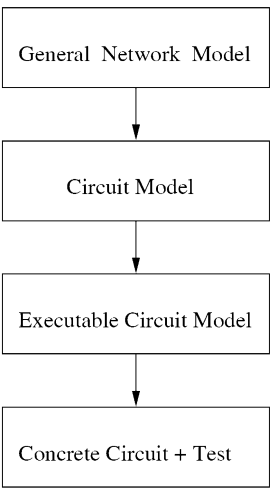

Figure 1. Development overview

Our work has been influenced by Dines Bjørner's TripTych dogma and formal techniques for software development described in Refs.[1, 6, 7]. We have been strongly inspired by the domain model for railway networks in Ref.[8]. The railway networks have many similarities with circuits that can also be considered as a kind of network

(called connectors in the railway model). The components provide internal connections between their connectors. The internal connections may change over time. For instance, in a circuit a contact may or may not provide a connection between its two ends, depending on whether the contact is pushed or not. Therefore, we got the idea first to make a common re-usable abstraction for all kinds of networks, and then refine this to a model for circuits. A development overview is given in Fig. 1.

Paper overview First, in Section 2, we will informally explain about relay circuits and their electrical behaviour. Next, in Sections 3–5, the formal domain model is stepwise developed from an abstract, property-oriented specification into an explicit, model-oriented specification that can be automatically translated into SML. Then, in Section 6, the model is instantiated for a concrete circuit, and it is shown how the instantiated model can be tested and executed. Finally, in Sections 7–8, future work is suggested and some conclusions are drawn. In appendix A, there is a short introduction to the RAISE method and its associated specification language RSL.

2 Relay Circuits and Diagrams

2.1 Relay circuits

A relay circuit is made up of components such as a power supply, relays, contacts, buttons, and wires.

A *relay* is an electrical switch operated by an electromagnet to connect or disconnect a number of contacts in a circuit. When current goes through the relay, the magnet is *drawn* and some of the associated contacts are connected (these contacts are said to be *upper contacts*) while others (the *lower contacts*) are disconnected. When no current goes through the relay, the magnet is *dropped* and the associated upper and lower contacts will be disconnected and connected, respectively. When contacts are connected/disconnected this may imply that sub-circuits containing these contacts become live/dead. This again may imply that relays of these sub-circuits are drawn or dropped.

Relay circuits implementing interlocking systems can get input from the environment:

- buttons can be pushed (and later released) by an operator
- track relays are dropped/drawn when trains enter/leave track sections
- point relays are dropped/drawn when points are moved into a new position

The two latter kinds of external events can, from a conceptual point of view, be considered as push and release events for buttons that control current through the track relays and point relays. In this paper we will take this view. Hence, the only external events are button events, and all relay events are considered as internal events.

An input may lead to a chain of internal events: relays that are drawn and dropped. For a well designed relay circuit such a chain of internal events will terminate after a few steps and take almost no time. Therefore, in this work we will assume

2.2 Diagrams

The Danish railways use diagrams to document the electrical circuits of a relay system. For each relay one of the diagrams shows the sub-circuit that controls that relay. The diagram visualises the initial state of the sub-circuit. An example of such a diagram is shown in Figure 2. This diagram shows the sub-circuit controlling a relay named $RR1$. The circuit consists of a number of components connected by wires. The wires are depicted as black lines. At the top is the positive pole and at the bottom is the negative pole of the power supply. Relay $RR1$ is shown using this signature:

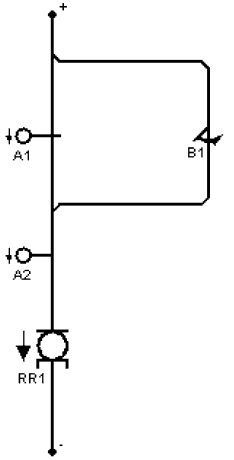

Figure 2. Diagram for circuit controlling relay $RR1$

Signatures for relays contain a circle that is decorated with some bars. The bars only indicate the logical purpose of the relay – they do not indicate any special physical behaviour. The downwards arrow informs that in the initial state this relay is dropped. (If it had been drawn the arrow would have been upwards.) A number of contacts belonging to other relays occur in this circuit. E.g. a contact belonging to a relay named $A1$ is shown using this signature:

The downwards arrow informs that in the initial state relay $A1$ is dropped. The horizontal bar breaks the wire – this indicates that the contact is disconnected in the initial state. As the contact is disconnected when the associated relay $A1$ is dropped, it is an upper contact of relay $A1$. If the bar had not been breaking the wire, as for $A2$, it would have indicated that the contact had been connected in the initial state. Also a button $B1$ is shown on the diagram using this signature:

This signature informs that in the initial state this button is released. A pushed

The connection points of a component are called *pins* and are given numbers. For a relay, the pin numbers will typically be 01 and 02. Sometimes the pin numbers of a component are shown on a diagram.

2.3 Electrical behaviour

In this section we present an example of the dynamics of another circuit. The example shows a scenario where a button of a circuit is pushed. In Fig. 3 the first four states of the circuit in this scenario are visualised on a diagram of the circuit. Wires that are current carrying are shown by a green[1] colour. State 0 is the initial state. In the initial state, no wires are current carrying. When the button B is pushed, current is flowing from plus to minus through relay $R1$, see state 1. As a consequence of this, relay $R1$ is drawn and its two associated upper contacts become connected, opening a second path of current from plus to minus through relay $R2$, see state 2. As current is flowing through relay $R2$, this will be drawn, see state 3. State 3 is *stable*, i.e. no more internal events can happen. If now button B was released in state 3, $R1$ would

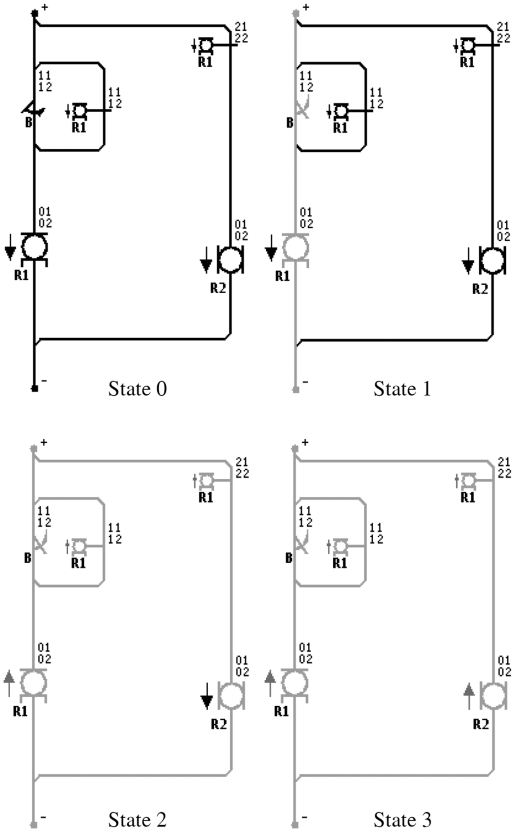

Figure 3. A state sequence for a circuit

stay drawn. The purpose of the loop involving the contact between pins 11 and 12 of $R1$ is exactly to achieve this.

3 General Network Model

In Ref.[8], Bjørner et. al. present an abstract model of railway networks. Parts of this model are specific for railways, but some parts can also be used for other kinds of network. In this section, we will extract the general parts of their model providing a general network model.

The idea behind this model is to consider a *network* as consisting of *components*[2] that are glued together with connectors. In the railway domain components are various kinds of track segments, and in the circuit domain components are electrical components such as relays, contacts, buttons, and wires. Components have associated *connectors* that can be considered as places where the components are glued together. Two components are connected if they share a connector.

Each component provides a collection of internal *connections*[3] between pairs of its own connectors. Hence connections go between connectors inside a component and not between components. The connections of a component may change over time. To be most general, we have chosen connections to be directed, i.e. there might be a connection from a connector $c1$ to another connector $c2$, but no connection from $c2$ to $c1$. In the railway domain a direction is needed in order to express that in certain cases it is only possible for a train to go in one direction through a track section. For the circuit domain such a distinction would not have been necessary as current can always flow in both directions through a component.

3.1 Networks, components and connectors

In the model we introduce abstract types for networks, components, and connectors, and concrete types for connections and sets of connections. We have chosen to use abstract types for those types that are expected to be refined into distinct concrete types for different domains. Hence we only use concrete types for types that should be the same in all refinements.

type

Network, Component, Connector,

Connection = Connector × Connector,

Connections = Connection**-set,**

We then introduce observer functions for the abstract types and specify axioms they must satisfy:

value

components : Network → Component**-set**,

connectors : Component → Connector**-set**,

[2] In Ref.[8] components are called units. To avoid confusion with the RSL reserved word **Unit**, in

```
    possibleConnections : Component → Connections-set,
    actualConnections : Component → Connections
  axiom
    [connectors_ok]
      ∀ cm : Component •
        (∀ s : Connections • s • possibleConnections(cm) ⇒
          (∀ (c1, c2) : Connection • (c1, c2) ∈ s  ⇒ {c1, c2} ⊆ connectors(cm))
        ),
    [actualConnections_are_possible]
      ∀ cm : Component • actualConnections(cm) ∈ possibleConnections(cm)
```

The observers express that a network has components, and a component has connectors, an actual connection set, and a set of physically possible connection sets.

The first axiom expresses that the connectors of each possible connection of a component are connectors of the component. The second axiom expresses that the actual connection set is a physically possible connection set.

Below are additional functions that we will need later on. The functions are all specified in terms of the basic observers declared above. Some of the functions are exactly the same modulo naming as in Ref.[8], while others are either totally new or have equivalent, but different explicit definitions. The first function tests whether there is a connection from one connector to another connector in a network. The remaining functions compute (1) all connectors of a network, (2) all actual connections of a network, (3) the set of network components that have a given connector, and (4) the set of all connectors $c2$ for which there is a connection from a given connector c to $c2$, respectively.

```
  value /∗ derived aux functions ∗/
    areConnected : Connector × Connector × Network → Bool
    areConnected(c1, c2, n) ≡
      (∃ cm : Component •
        cm ∈ components(n) ∧ (c1, c2) ∈ actualConnections(cm)),

    connectors : Network → Connector-set
    connectors(n) ≡
      dunion({connectors(cm) | cm : Component • cm ∈ components(n)}),

    allConnections : Network → Connection-set
    allConnections(n) ≡
      dunion({actualConnections(cm) | cm : Component • cm ∈ components(n)}),

    components :  Connector × Network → Component-set
```

```
    {cm | cm : Component • cm ∈ components(n) ∧ cn ∈ connectors(cm) },

  connected_to : Connector × Network → Connector-set
  connected_to(c, n) ≡
    {c2 | c2 : Connector • c2 ∈ connectors(n) ∧ areConnected(c, c2, n)
```

where dunion is distributed union:

```
  dunion({s1, ..., sn}) = s1 ∪ ... ∪ sn
```

3.2 Paths

In the model we introduce the notion of a *path*[4] and a number of useful functions concerning paths. A path is modelled as a list of connectors:

```
  type Path = Connector*
```

and we introduce a function *available* that can check whether a path is available in a network:

```
value
  available : Path × Network → Bool
axiom
  [available_req]
    ∀ p : Path, n : Network •
      available(p, n) ⇒
        len p ⩾ 2 ∧
        (∀ i : Nat • i ∈ {1 .. len p − 1} ⇒ areConnected(p(i), p(i + 1), n))
```

The axiom expresses that when a path is available in a network, (1) it contains at least two connectors, and (2) any consecutive connectors $c1$ and $c2$ in the path are connected in the network, i.e. there is a component in the network for which $(c1, c2)$ is an actual connection. There may be additional requirements for a path to be available, but these may differ from domain to domain. We therefore use an implication in the axiom.

Here is a function that returns the set of all available paths from one connector to another connector in a network:

```
value
  availablePaths : Connector × Connector × Network → Path-set
  availablePaths(from, to, n) ≡
    { p | p : Path • available(p, n) ∧ p(1) = from ∧ p(len p) = to }
```

Another useful function is one that can test whether a path contains a given connection:

value
PathContainsConnection : Path × Connection → **Bool**
PathContainsConnection(p, (c1, c2)) ≡
(∃ i : **Nat** • i ∈ {1 .. **len** p − 1 } ∧ (p(i), p(i+1)) = (c1, c2))

3.3 Relation to graphs

The network model we have presented above can be seen as a refinement of an abstract graph model that includes the following types and observes

type Graph, Node, Edge = Node × Node
value
nodes : Graph → Node-**set**,
edges : Graph → Edge-**set**

and a number of useful derived functions that are relevant for graphs. The types *Network*, *Connector*, and *Connection* refine *Graph*, *Node* and *Edge*, respectively. The functions *connectors* and *allConnections* refine *nodes* and *edges*, respectively. Several of our derived functions are refinements of corresponding functions for graphs.

4 Circuit Model

In this section we show how the general network model can be refined into a circuit model. We only explain those parts that are changed or new.

4.1 Connectors

In a circuit there are two distinguished connectors: *pos* and *neg* representing the positive and the negative pole of the power supply. We add declarations of these:

value
pos : Connector,
neg : Connector
axiom pos ≠ neg

4.2 Components

In the circuits we are modelling in this paper there are four kinds of components: relays, contacts, buttons, and wires.

We introduce specialised types *Relay*, *Contact*, *Button*, *Wire* for each of these kinds of components, and we define specialised *connectors*, *possibleConnections* and *actualConnections* functions for each of these types. These types and functions will be presented in subsections of this section.

```
type
  Component == Component_from_Relay(r : Relay) |
               Component_from_Contact(c : Contact) |
               Component_from_Button(b : Button) |
               Component_from_Wire(w : Wire)
```

and the axiomatic specification of the *possibleConnections*, *actualConnections* and *connectors* functions for *Component*s is replaced with explicit function definitions that make cases over the specialised functions. E.g.

```
value
  connectors : Component → Connector-set
  connectors(cm) =
    case cm of
      Component_from_Relay(r) → connectors(r),
      Component_from_Contact(c) → connectors(c),
      Component_from_Button(b) → connectors(b),
      Component_from_Wire(w) → connectors(w)
    end
```

The two axioms [connectors_ok] and [actualConnections_are_possible] shown in Section 3.1 for the general network model can be proved to hold in the circuit model. This is done by a case analysis over the kinds of components.

For each kind of component we add a function that can test whether a given component is of that kind. E.g.:

```
value
  isWire : Component → Bool
  isWire(cm) ≡
    case cm of
      Component_from_Wire(_) → true,
      _ → false
    end
```

4.2.1 Component identifiers

We introduce abstract types of identifiers for relays, contacts, buttons, and wires:

```
type RelayId, ContactId, ButtonId, WireId
```

4.2.2 Relays

A relay has the following static properties: it has two distinct connectors, a set of upper contacts, and a set of lower contacts. The sets of upper and lower contacts must be disjoint. The relay has one basic, dynamic property: it may be drawn or not.

We define the *Relay* type as a record type having fields for its static and dynamic

```
type Relay ::
  connector1 : Connector /∗ static ∗/
  connector2 : Connector /∗ static ∗/
  upperContacts : ContactId-set /∗ static ∗/
  lowerContacts : ContactId-set  /∗ static ∗/
  drawn : Bool /∗ dynamic ∗/
```

and we introduce a wf function that can be used for checking a relay for consistency between its field values:

```
value
  wf : Relay → Bool
  wf(r) ≡
    connector1(r) ≠ connector2(r) ∧
    upperContacts(r) ∩ lowerContacts(r) = { }
```

The set of connectors of a relay consists of its two connectors:

```
value
  connectors :  Relay → Connector-set
  connectors(r) ≡ {connector1(r), connector2(r)}
```

A relay always provides connections in both directions between its two connectors, say $c1$ and $c2$. Therefore the only possible set of connections is $\{(c1, c2), (c2, c1)\}$, and the actual set of connections is that set. Hence, the functions *possibleConnections* and *actualConnections* are defined as follows:

```
value
  possibleConnections : Relay → Connections-set
  possibleConnections(r) ≡
    let c1 = connector1(r), c2 = connector2(r) in
      {{(c1,c2), (c2,c1)}}
    end,

  actualConnections : Relay → Connections
  actualConnections(r) = hd possibleConnections(r)
```

Note that the **hd** operator can be used to select an arbitrary member of a non-empty set. It is easy to see that the two component axioms [connectors_ok] and [actualConnections_are_possible] hold for relays.

4.2.3 Contacts

Like relays, a contact has two distinct connectors, say $c1$ and $c2$. A contact may be connected or not. When it is connected it provides connections between $c1$ and $c2$ in both directions. Hence the specialised type and functions for contacts are defined

```
type Contact ::
  connector1 : Connector /∗ static ∗/
  connector2 : Connector /∗ static ∗/
  connected : Bool ∗/ dynamic ∗/

value
  wf : Contact → Bool
  wf(c) ≡ connector1(c) ≠ connector2(c)

value
  connectors : Contact → Connector-set
  connectors(c) ≡ {connector1(c), connector2(c)}

value
  /∗ a contact either connects its two ends or nothing ∗/
  possibleConnections : Contact → Connections-set
  possibleConnections(c) ≡
    let c1 = connector1(c), c2 = connector2(c) in
      {{ }, {(c1,c2), (c2,c1)}}
    end,

  actualConnections : Contact → Connections
  actualConnections(c) ≡
    let c1 = connector1(c), c2 = connector2(c) in
      if connected(c) then {(c1,c2), (c2,c1)} else {} end
    end
```

It is easy to see that the two component axioms [connectors_ok] and [actualConnections_are_possible] hold for contacts.

4.2.4 Buttons

Like contacts, a button has two distinct connectors, say $c1$ and $c2$. A button may be pushed or not. When it is pushed it provides connections between $c1$ and $c2$ in both directions. Hence the specialised type and functions for buttons are defined just like for contacts except that the field name *connected* is replaced with *pushed*.

4.2.5 Wires

Like relays, a wire has two distinct connectors, and it always provide connections between these in both directions. It has no other basic attributes. Hence, the specialised type for wires is defined as follows:

```
type Wire ::
  connector1 : Connector /∗ static ∗/
```

The functions are defined as for relays, except for the wf function:

```
value
  wf : Wire → Bool
  wf(w) ≡ connector1(w) ≠ connector2(w)
```

4.3 Networks

We replace the abstract *Network* type with a record type that for each kind of component provides an association between identifiers and components of that kind:

```
type
  Network ::
    relays : RelayId m̄ Relay
    contacts : ContactId m̄ Contact
    buttons : ButtonId m̄ Button
    wires : WireId m̄ Wire
```

Then the *components* function can be given an explicit definition:

```
value
  components : Network → Component-set
  components(n) ≡
    let mk_Network(rm, cm, bm, wm) = n in
      {Component_from_Relay(rm(rid)) | rid : RelayId • rid ∈ dom rm}
      ∪
      {Component_from_Contact(cm(cid)) | cid : ContactId • cid ∈ dom cm}
      ∪
      {Component_from_Button(bm(bid)) | bid : ButtonId • bid ∈ dom bm}
      ∪
      {Component_from_Wire(wm(wid)) | wid : WireId • wid ∈ dom wm}
    end
```

4.3.1 Well-Formedness

Not all values of the *Network* type represent legal circuits. We therefore introduce a function, *legal* that can check this:

```
value
  legal : Network → Bool
  legal(n) ≡ ...
```

The right-hand side is a conjunction of all the required conditions. We will now explain and show these.

Each component of the network must be well-formed. For relays this condition

$$\forall\ r : \text{Relay} \bullet r \in \mathbf{rng}\ \text{relays}(n) \Rightarrow \text{wf}(r)$$

Furthermore, the information about relays and contacts must be consistent:

- Two relays must not share contacts:

$$\begin{array}{l}\forall\ r1 : \text{Relay} \bullet r1 \in \mathbf{rng}\ \text{relays}(n) \Rightarrow \\ \quad (\forall\ r2 : \text{Relay} \bullet r2 \in \mathbf{rng}\ \text{relays}(n) \land r1 \neq r2 \Rightarrow \\ \quad\quad (\text{upperContacts}(r1) \cup \text{lowerContacts}(r1)) \cap \\ \quad\quad (\text{upperContacts}(r2) \cup \text{lowerContacts}(r2)) = \{\} \\ \quad)\end{array}$$

- All contacts must belong to some relay:

$$\begin{array}{l}\forall\ cid : \text{ContactId} \bullet cid \in \mathbf{dom}\ \text{contacts}(n) \Rightarrow \\ \quad (\exists\ r : \text{Relay} \bullet r \in \mathbf{rng}\ \text{relays}(n) \land \\ \quad\quad cid \in \text{upperContacts}(r) \cup \text{lowerContacts}(r))\end{array}$$

- All upper/lower relay contacts must exist and have states consistent with their relay:

$$\begin{array}{l}\forall\ r : \text{Relay} \bullet r \in \mathbf{rng}\ \text{relays}(n) \Rightarrow \\ \quad (\forall\ cid : \text{ContactId} \in cid \in \text{upperContacts}(r) \Rightarrow \\ \quad\quad cid \in \mathbf{dom}\ \text{contacts}(n) \land \\ \quad\quad \text{connected}(\text{contacts}(n)(cid)) = \text{drawn}(r) \\ \quad) \land \\ \quad (\forall\ cid : \text{ContactId} \bullet cid \in \text{lowerContacts}(r) \Rightarrow \\ \quad\quad cid \in \mathbf{dom}\ \text{contacts}(n) \land \\ \quad\quad \text{connected}(\text{contacts}(n)(cid)) = \sim\text{drawn}(r) \\ \quad)\end{array}$$

Finally, components must share connectors in a legal way:

- There must be a positive and a negative pole:

$$\{\text{pos}, \text{neg}\} \subseteq \text{connectors}(n)$$

- Only wires can be connected to the poles:

$$\begin{array}{l}\forall\ cm : \text{Component} \bullet \\ \quad cm \in \text{components}(\text{pos}, n) \cup \text{components}(\text{neg}, n) \Rightarrow \text{isWire}(cm)\end{array}$$

$$\forall\ \text{cm1} : \text{Component} \bullet \text{cm1} \in \text{components(n)} \Rightarrow$$
$$(\forall\ \text{cm2} : \text{Component} \bullet \text{cm2} \in \text{components(n)} \wedge \text{cm1} \neq \text{cm2} \Rightarrow$$
$$\textbf{card}\ (\text{connectors(cm1)} \cap \text{connectors(cm2)}) > 0 \Rightarrow$$
$$\text{isWire(cm1)} \vee \text{isWire(cm2)}$$
$$)$$

- Two components share at most one connector:

$$\forall\ \text{cm1} : \text{Component} \bullet \text{cm1} \in \text{components(n)} \Rightarrow$$
$$(\forall\ \text{cm2} : \text{Component} \bullet \text{cm2} \in \text{components(n)} \wedge \text{cm1} \neq \text{cm2} \Rightarrow$$
$$\textbf{card}\ (\text{connectors(cm1)} \cap \text{connectors(cm2)}) \leqslant 1$$
$$)$$

  This ensures that a wire can't bypass another component (which would mean that the component had no function in the circuit). The condition also means that two connectors at most designate one component – a property that we utilize when defining the dynamic behaviour of circuits in Section 4.5.

- A connector must connect at least 2 and at most 3 components, unless it is a pole:

$$\forall\ \text{cn} : \text{Connector} \bullet$$
$$\text{cn} \in \text{connectors(n)} \setminus \{\text{pos, neg}\} \Rightarrow \textbf{card}\ \text{components(cn, n)} \in \{2,3\}$$

  This and the other conditions ensure that the following rules required by BaneDanmark for the Danish relay circuits are fulfilled:

  1. Each relay/contact/button connector must have 1 or 2 wires attached.
  2. Each connector of a wire must be attached to 1 or 2 other components, unless the connector is a pole.

  The lower bound is used to ensure that components are not partially connected, and the upper bound is used to ensure that the electrical connections between components is stable (if too many wires are connected to the same component, some of them may fall down).

Note that the rules for legal connector sharing are concerned with completed relay circuits and not with partially constructed relay circuits (having components only partially connected to other components) or relay circuits containing components only to be used in anticipated extensions (in which case such components should be bypassed by wires until the extensions are made). If one wish to check incomplete circuits, the requirements should be relaxed.

4.4 Paths

We replace the axiomatic definition of the *available* function with an explicit

value

available : Path × Network → **Bool**

available(p, n) ≡

  len p ⩾ 2 ∧

  (∀ i : **Nat** • i ∈ {1} **len** p − 1} ⇒ areConnected(p(i), p(i+1), n)) ∧

  (∀ i, j : **Nat** • {i,j} ⊆ {1 .. **len** p} ⇒ (i ≠ j ⇒ p(i) ≠ p(j)))

For a path to be available in a circuit there is one more requirement than in the general model: there must be no loops or reversed directions (asserted by the last conjunct, which ensures no repetitions of connectors in a path). It is easy to see that the definition of *available* satisfies the [*available_req*] axiom shown in Section 3.2.

4.5 Events

We now extend our circuit model with functions that model the dynamic evolution of circuits. In a circuit the following kinds of external events can happen:

- A button is pushed.
- A button is released.

and the following kinds of internal events can happen:

- A relay is drawn and its upper/lower contacts are connected/disconnected.
- A relay is dropped and its upper/lower contacts are disconnected/connected.

For each kind of event, we declare a *state generator function*[5] having a signature of the form:

value gen : . . . × Network $\xrightarrow{\sim}$ Network

and an associated *guard* having the signature:

value can_gen : . . . × Network → **Bool**

The *gen* function defines the associated state changes.

4.5.1 External events

For the button events, we have the following guards expressing that a button can be pushed/released, if it is not already pushed/released and the network is stable, i.e. no more internal (relay) events can happen, cf. the assumption we made in Section 2.1:

```
value
  /∗ guards for external events ∗/
  can_push : ButtonId × Network → Bool
  can_push(bid, n) ≡
    bid ∈ dom buttons(n)  ∧
    is_stable(n) ∧
    let b = buttons(n)(bid) in ∼pushed(b) end,

  can_release : ButtonId × Network → Bool
  can_release(bid, n) ≡
    bid ∈ dom buttons(n)  ∧
    is_stable(n) ∧
    let b = buttons(n)(bid) in pushed(b) end,

  /∗ aux function ∗/
  is_stable : Network → Bool
  is_stable(n) ≡
    (∀ rid : RelayId • rid ∈ dom relays(n) ⇒
      (∼ can_draw(rid, n)) ∧  (∼ can_drop(rid, n))
    )
```

The generator for pushing a button is defined as follows:

```
value
  push : ButtonId × Network ~→ Network
  push(bid,n) ≡
    let
      mk_Network(rm, cm, bm, wm) = n
    in
      mk_Network(rm, cm, bm † [bid ↦ push(bm(bid))], wm)
    end
  pre can_push(bid, n),
```

This function uses the following generator function that is added to the specification of buttons given in Section 4.2.4:

```
  push : Button → Button
  push(b) ≡
    let mk_Button(c1, c2, p) = b : Button in
      mk_Button(c1, c2, true)
    end
```

4.5.2 Internal events

For the first kind of internal event we have a guard expressing that a relay with identifier *rid* can be drawn if the relay is not already drawn and one of its connections is current carrying (what this means is explained further below):

```
value
  /∗ guards for internal events ∗/
  can_draw : RelayId × Network → Bool
  can_draw(rid, n) ≡
    rid ∈ dom relays(n) ∧
    let r = relays(n)(rid) in
      ∼ drawn(r) ∧
      (∃ cn : Connection • cn ∈ actualConnections(r) ∧
          currentCarrying(cn, n)
      )
    end
```

Likewise, for the second kind of internal event, we have a guard expressing that a relay with identifier *rid* can be dropped if the relay is drawn (i.e. not already dropped) and none of its connections are current carrying:

```
  can_drop : RelayId × Network ~→ Bool
  can_drop(rid, n) ≡
    rid ∈ dom relays(n) ∧
    let r = relays(n)(rid) in
      drawn(r) ∧
      ∼ (∃ cn : Connection • cn ∈ actualConnections(r) ∧
          currentCarrying(cn, n)
        )
    end
```

A connection is current carrying, if there exists an available path from the positive pole to the negative pole in the network such that it contains that connection:

```
  currentCarrying : Connection × Network → Bool
  currentCarrying(con, n) ≡
    (∃ p : Path • p ∈ availablePaths(pos, neg, n) ∧
                                  PathContainsConnection(p, con))
```

```
draw : RelayId × Network ~→ Network
draw(rid, n) ≡
  let
    mk_Network(rm, cm, bm, wm) = n,
    r = rm(rid),
    ucs = upperContacts(r),
    lcs = lowerContacts(r)
  in
    mk_Network(
      rm † [rid ↦ draw(r)],
      cm † [cid ↦ connect(cm(cid)) | cid : ContactId • cid ∈ ucs]
           † [cid ↦ disconnect(cm(cid)) | cid : ContactId • cid ∈ lcs],
      bm,
      wm)
  end
pre can_draw(rid, n),
```

This function uses the following generator functions that are added to the specifications of relays and contacts given in Sections 4.2.2 and 4.2.3, respectively:

```
draw : Relay → Relay
draw(r) ≡
  let mk_Relay(c1, c2, u, l, d) = r : Relay in
    mk_Relay(c1, c2, u, l, true)
  end,

connect : Contact → Contact
connect(c) ≡
  let mk_Contact(c1, c2, b) = c : Contact in
    mk_Contact(c1, c2, true)
  end,

disconnect : Contact → Contact
disconnect(c) ≡
  let mk_Contact(c1, c2, b) = c : Contact in
    mk_Contact(c1, c2, false)
  end
```

A generator *drop* for dropping a relay is defined similarly.

In the rest of this section, we will discuss how to use the functions introduced above and we will introduce additional functions that can be used for exploring cir-

Guarded applications of the *draw* and *drop* generators can be used to find the next states of a circuit. Examples of this will be shown in Section 6.

A circuit might be in a state where more than one relay can be drawn or dropped. We introduce functions that return the sets of relay identifiers of relays that can be drawn and dropped for a given circuit (i.e. *Network* value):

```
can_be_drawn : Network → RelayId-set
can_be_drawn(n) ≡
  {rid | rid : RelayId • rid ∈ dom relays(n) ∧ can_draw(rid, n)}
```

can_be_dropped is defined similarly. Having these functions, we can define a function that returns the number of relays of a network that can be drawn or dropped:

```
num_enabled_relays : Network → Nat
num_enabled_relays(n) ≡
  card can_be_drawn(n) + card can_be_dropped(n) = 1,
```

When a circuit is in a state where more than one relay event is possible, then there are several possibilities for the next event to happen. For instance, if the circuit is in a state where both relay $r1$ and relay $r2$ can be drawn, then the next event will either be that $r1$ is drawn or that $r2$ is drawn. In such a case there is the risk that the next stable state depends on which of the events happens first. This would for instance be the case, if drawing $r1$ leads to a stable state such that $r2$ can't be drawn, and drawing $r2$ leads to a stable state where $r1$ can't be drawn. Such a case is called a *race condition* and we say that the circuit is *non-deterministic*.

In the synchronous approach to modelling[9] of reactive systems one would in the case of several possible events take the simplifying assumption that all events happen simultaneously. This has the advantage that one can define a deterministic next-state function, but the disadvantage that this may give wrong results for circuits having race conditions. Below we introduce such a function, called *next*. It takes a network n and produces a new network by *simultaneously* drawing all relays r that can be drawn in n and dropping all relays r that can be dropped in n:

```
next : Network ~→ Network
next(n) ≡
  let
    mk_Network(rm, cm, bm, wm) = n,
    drawn_relays =
      [rid ↦ draw(rm(rid)) | rid : RelayId • rid ∈ dom rm ∧
        can_draw(rid,n)],
    dropped_relays =
      [rid ↦ drop(rm(rid)) | rid : RelayId • rid ∈ dom rm ∧
        can_drop(rid,n)],
    connected_contacts =
      [cid ↦ connect(cm(cid)) | cid : ContactId • cid ∈ dom cm ∧
```

```
            ((can_draw(rid,n) ∧ cid ∈ upperContacts(rm(rid))) ∨
              (can_drop(rid,n) ∧ cid ∈ lowerContacts(rm(rid)))
          ) )
        ],
      disconnected_contacts =
        [cid ↦ disconnect(cm(cid)) | cid : ContactId • cid ∈ dom cm ∧
          (∃ rid : RelayId • rid ∈ dom rm ∧
            ((can_draw(rid,n) ∧ cid ∈ lowerContacts(rm(rid))) ∨
              (can_drop(rid,n) ∧ cid ∈ upperContacts(rm(rid)))
          ) )
        ]
    in
      mk_Network(
        rm † (drawn_relays ∪ dropped_relays),
        cm † (connected_contacts ∪ disconnected_contacts),
        bm,
        wm)
    end
  pre legal(n)
```

Note that the union of the maps *drawn_relays* and *dropped_relays* is well-defined as the domains of these maps are disjoint[6] as $can\_draw(rid, n)$ and $can\_drop(rid, n)$ can't be true at the same time. The maps *connected_contacts* and *disconnected_contacts* and their union are also well-defined as for legal networks each contact belongs at most to one relay and the upper and lower contacts of a relay are disjoint.

As mentioned earlier, the function *next* assumes there are no race conditions if it is to accurately model circuit behaviour. Its use in simulating circuits (by repeated application of *next* until stability, no more internal events, is reached) will assume eventual stability. We return to how we can check for race conditions and eventual stability in Section 7.

5 Refinement of Circuit Model

In this section we refine the circuit model to make it translatable into SML.

5.1 Refinement of types

Abstract types are not translatable, so the abstract types are refined to concrete types:

```
type
  Connector = Text,
  RelayId = Text,
```

ContactId = **Text**,
ButtonId = **Text**,
WireId = **Text**

and the constants *pos* and *neg* are explicitly defined:

value
pos : Connector = ″+″, neg : Connector = ″−″

For this to be a refinement, the axiom $pos \neq neg$ of the previous model must be satisfied. This is clearly the case.

5.2 Refinement of functions

Only two of the function definitions are not translatable and should be refined.

5.2.1 The available function

The definition of the *available* function is not translatable as it contains a quantified expression that is not translatable:

$\forall$ i, j : **Nat** • {i,j} $\subseteq$ {1 .. **len** p} $\Rightarrow$ (i $\neq$ j $\Rightarrow$ p(i) $\neq$ p(j))

For a universal quantified expression to be translatable, it must be in the form

$\forall$ id : T • id $\in$ s $\Rightarrow$ eb

where s is a finite set. We therefore replace the non translatable expression with the following equivalent expression that is translatable:

$\forall$ i : **Nat** • i $\in$ {1 .. **len** p} $\Rightarrow$
($\forall$ j : **Nat** • j $\in$ {1 .. **len** p} $\Rightarrow$ (i $\neq$ j $\Rightarrow$ p(i) $\neq$ p(j)))

5.2.2 The availablePaths function

The definition of the *availablePaths* function is not translatable as it contains a set comprehension that is not translatable. We therefore replace it with the following translatable definition:

value
availablePaths : Connector × Connector × Network → Path-**set**
availablePaths(from, to, n) ≡
if from = to **then** {} **else** extend_path(⟨ from ⟩, to, n) **end**

that uses two mutually recursive auxiliary functions:

extend_path : Path × Connector × Network → Path-**set**
extend_path(p, to, n) ≡

```
      if c1 = to then {p}
        else
          let tobevisited = connected_to(c1, n)\(elems p) in
            extend_path_with(p, to, tobevisited, n)
          end
      end
    end
  pre len p > 0,

  extend_path_with : Path × Connector × Connector-set × Network → Path-set
  extend_path_with(p, to, tobevisited, n) ≡
    if tobevisited = {} then {}
    else
      let c = hd tobevisited in
        extend_path(p ^ ⟨c⟩, to, n)
        ∪
        extend_path_with(p, to, tobevisited \ {c}, n)
      end
    end
  pre len p > 0
```

Assume that p is an available path $p1^\frown\langle c1\rangle$ in n. Then $extend\_path(p, to, n)$ returns the set of all extensions $p^\frown p'$ of this path such that the extensions are available paths in n ending at to. It works as follows: If $c1 = to$ then p is an available path that ends at to and $\{p\}$ is returned. Otherwise it calculates a set $tobevisited$ of those connectors to which there is a connection from $c1$ and which are not already in p, and then it makes the call $extend\_path\_with(p, to, tobevisited, n)$, the effect of which is explained below.

Assume that p is a non-empty path $p1^\frown\langle c1\rangle$ and $tobevisited$ is a set of all connectors c not already present in p and for which there is a connection from $c1$ to c in n. Then $extend\_path\_with(p, to, tobevisited, n)$ calculates for each c in $tobevisited$ the set $extend\_path(p^\frown\langle c\rangle, to, n)$ of all available paths in n that extend $p^\frown\langle c\rangle$ and end at to, and takes the union of these sets of paths.

Hence, the set of all available paths from a connector $from$ to a connector to can be found by the expression $extend\_path(\langle from\rangle, to, n)$.

It should be proved that the new and old definition of $availablePaths$ are equivalent, i.e.

```
∀ from, to : Connector, n : Network •
  if from = to then {} else extend_path(⟨ from ⟩, to, n) end ≡
    {p | p : Path • available(p, n) ∧ p(1) = from ∧ p(len p) = to}
```

6 Concrete Circuits and Testing

In this section we will show by an example how a concrete circuit c can be defined as a value of type $Network$, and how test cases can be expressed at specification level and executed for the circuit.

6.1 Circuit specification

We will now show how the circuits shown in Fig. 3 can be modelled as values $c0$, $c1$, $c2$, and $c3$ of type $Network$:

```
value
  c0 : Network =
    mk_Network(
    [″R1″ ↦ mk_Relay(″ R1.01 ″, ″ R1.02 ″, {″C1″, ″C2″}, {}, false),
    ″R2″ ↦ mk_Relay(″ R2.01 ″, ″ R2.02 ″, {}, {}, false)],
    [″C1″ ↦ mk_Contact (″ R1.11 ″, ″ R1.12 ″, false),
    ″C2″ ↦ mk_Contact (″ R1.21 ″, ″ R1.22 ″, false)],
    [″B″ ↦ mk_Button (″ B.11 ″, ″ B.12 ″, false)],
    [″W1″ ↦ mk_Wire (″ + ″, ″ B.11 ″),
    ″W2″ ↦ mk_Wire (″ + ″, ″ R1.21 ″),
    ″W3″ ↦ mk_Wire (″ B.11 ″, ″ R1.11 ″),
    ″W4″ ↦ mk_Wire (″ B.12 ″, ″ R1.01 ″),
    ″W5″ ↦ mk_Wire (″ B.12 ″, ″ R1.12 ″),
    ″W6″ ↦ mk_Wire (″ R1.22 ″, ″ R2.01 ″),
    ″W7″ ↦ mk_Wire (″ R1.02 ″, ″ - ″),
    ″W8″ ↦ mk_Wire (″ R2.02 ″, ″ - ″)]
  )
```

The values $c1$, $c2$ and $c3$ are defined similarly.

6.2 Testing

A specification that is completely concrete can often be translated automatically to a programming language and executed. The RAISE tools have translators to e.g. SML [10] and to C++. There is also an extension to RSL providing **test_case** declarations. A **test_case** declaration consists of a sequence of test cases with a syntax similar to that for axioms: an optional identifier plus an expression. The expression can be of any type. Test cases do not affect the semantics of specifications, but the translators will generate code to evaluate them and print the identifier and result values.

One can design systematic functional and structural tests for concrete specifications just as for programs[11], and the RAISE tools will give information on the test coverage. Test cases may also be derived from axioms of the abstract specifications that the concrete specification refines. E.g. one can make the following test case

model in Section 3 holds for all components of the $c0$ circuit:

test_case
[actualConnections_are_possible_in_c0]
∀ cm : Component · cm ∈ components(c0) ⇒
actualConnections(cm) ∈ possibleConnections(cm)

When running this test case with the RAISE tools, the result is:

[actualConnections_are_possible_in_c0] **true**

showing that the property is true. However, when all refinement steps are formally proved, it is not strictly necessary to make such checks of axioms.

As systematic test is a well-known topic, we will not discuss this further in this paper, but show how one can execute test cases to explore models of circuits.

6.3 Investigating circuit models

For the circuits $c0$, $c1$, $c2$, and $c3$ one can write tests cases to check whether they are legal. For $c0$ the test case looks like this:

test_case
[legal_c0] legal(c0)

When running this test case with the RAISE tools, the result is:

[legal_c0] **true**

showing that $legal(c0)$ is **true**.

The dynamic behaviour of circuits can be simulated using guarded applications of the *Network* generator functions introduced in Section 4.5. The following test cases express that the circuit scenario $c0$, $c1$, $c2$, and $c3$ visualised in Figure 3 can be achieved by pushing B in the initial circuit $c0$, then drawing relay $R1$ and finally drawing relay $R2$:

test_case
[can_push_B_in_c0] can_push(″B″, c0),
[push_B_in_c0] c1 = push(″B″, c0),
[can_draw_R1_in_c1] can_draw(″R1″, c1),
[draw_R1_in_c1] c2 = draw(″R1″, c1),
[can_draw_R2_in_c2] can_draw(″R2″, c2),
[draw_R2_in_c2] c3 = draw(″R2″, c2),
[is_stable_c3] is_stable(c3)

The last test case checks that circuit $c3$ is stable. All these test cases give **true** when executed with the RAISE tools.

Alternatively to providing the expected results in the test cases, one could just let the test cases produce the new circuits or some observables of the new circuits and in this way get a simulation. Below we have done the latter where the obsery-

that give the sets of drawn relays, closed contacts, pushed buttons, and live wires of a circuit. To produce the circuit values $c2$ and $c3$, we could have used guarded applications of the *Network* generators, but that requires an insight into which events are possible, so instead we have chosen to use the synchronous *next* function introduced in Section 4.5.2.

value
c0 : Network = ... as before ...,
c1 : Network = push(B, c0),
c2 : Network = next(c1),
c3 : Network = next(c2)

test_case
"Testing c0:",
drawnRelays(c0),
closedContacts(c0),
pushedButtons(c0),
liveWires(c0),

"Testing c1:",
drawnRelays(c1),
closedContacts(c1),
pushedButtons(c1),
liveWires(c1),

"Testing c2:",
drawnRelays(c2),
closedContacts(c2),
pushedButtons(c2),
liveWires(c2),

"Testing c3:",
drawnRelays(c3),
closedContacts(c3),
pushedButtons(c3),
liveWires(c3)

When running these test cases the output is:

"Testing c0:"
{}
{}
{}

″Testing c1:″
{}
{}
{B}
{W7,W4,W1}
″Testing c2:″
{R1}
{C2, C1}
{B}
{W8,W7,W6,W5,W4,W3,W2,W1}
″Testing c3:″
{R1, R2}
{C2, C1}
{B}
{W8,W7,W6,W5,W4,W3,W2,W1}

When running the following test cases

test_case
[num_enabled_relays_c0] num_enabled_relays(c0),
[num_enabled_relays_c1] num_enabled_relays(c1),
[num_enabled_relays_c2] num_enabled_relays(c2),
[num_enabled_relays_c3] num_enabled_relays(c3)

we get the output:

[num_enabled_relays_c0] 0
[num_enabled_relays_c1] 1
[num_enabled_relays_c2] 1
[num_enabled_relays_c3] 0

which shows that in $c0$ and $c3$ no relay events are possible (i.e. they are stable), while in $c1$ and $c2$ there were exactly one possible relay event. The latter means that the function calls $next(c1)$ and $next(c2)$ each only performed one relay event and therefore gave the only possible next state of $c1$ and $c2$, respectively.

7 Future Work on Property Checking

As mentioned in Section 2.1, an input may lead to a chain of internal events: relays that are drawn and dropped. This raises two questions. The first question is whether such a chain would *terminate*, i.e. whether there is always a future state that is stable so that no more internal (relay) events can happen before a new input happens. Another question is whether the circuit behaviour is *deterministic*/has no race conditions, i.e. that for the same input, it will always end up in the same stable

for the use of the *next* function to model circuit behaviour accurately, and termination to make simulation possible. Note that the two properties are not just properties that enable modelling, they are also, in general, desirable properties of circuits. In future work it should be investigated how the questions above can be answered. Here we present some initial thoughts.

If the two questions should be answered in general for *any* reachable state and any input, an approach could be to use model checking.

If one should only investigate for a *single* state whether there exists a unique next stable state, one could do that by testing if there was a *nextStableState* function that took a state and returned a set of all possible next stable states that could be achieved by different interleavings of guarded applications of the relay event generators. Determinism could then be checked by investigating whether the returned set only contained one state. To avoid an infinite loop in case of a non-terminating chain, the function could count the number of events in each chain under investigation. If the count at a point should become larger than the possible number of relay states (which is 2^n, where n is the number of relays) then the function should return some error value to indicate that at least one chain did not terminate. The use of such a count would only be feasible for n being small, but already for small stations n is large (For a small station as Stenstrup in Denmark it was about 60). However, for relay circuits used in railway interlocking systems a chain of relay events should not only terminate, it should also be very short. Typically, there will be 0–5 events in a chain. So instead of stopping when the count is 2^n, one could already stop when the count becomes greater than some maximal allowed length l_max that is chosen much smaller than 2^n. If all the considered chains satisfy the requirement about being short, they are also known to terminate.

8 Conclusions

In this paper we have (1) provided a general network model, (2) refined this model to a domain model for relay circuits, and (3) shown how the domain model can be instantiated for a concrete circuit, and the instantiated model can be tested and executed. The method for deriving a domain model for relay circuits from the general network model can also be used for making other specialised network models.

In future work we would like to experiment with other domain models for relay circuits and compare them. Furthermore, tools for automated analyses of circuits should be developed in the context of such a domain model. The tools should comprise a simulator that can visualise the dynamics of relay circuits, and tool support for formal verification. A prototype of a graphical simulator has already been developed in Ref.[12] (and used for the production of circuit diagrams in this paper), but this tool was not developed from a formal domain model. Some experiments on model checking an RSL-SAL model of a relay interlocking system for a Danish station have been made in Ref.[13]. It was successfully checked that the system had some safety properties and that it will always terminate without any race conditions.

Acknowledgements

The author would like to thank (1) Kirsten Mark Hansen, Banedanmark, for

UNU/IIST, and the anonymous referees for very valuable comments to a draft version of this paper, and (3) Andreas Andersen Kjær and Marie Le Bliguet for help with production of figures.

References

[1] Bjørner D. Software Engineering, vol. 3: Domains, Requirements and Software Design. Texts in Theoretical Computer Science. Springer, 2006.
[2] The RAISE Language Group. The RAISE Specification Language. The BCS Practitioners Series. Prentice Hall Int., 1992.
[3] The RAISE Method Group. The RAISE Development Method. The BCS Practitioners Series. Prentice Hall Int., 1995.
[4] George C, Haxthausen AE. The Logic of the RAISE Specification Language. In Logics of Specification Languages, EATCS. Springer, 2008.
[5] European Committee for Electrotechnical Standardization. EN 50128 – Railway applications Communications, signalling and processing systems – Software for railway control and protection systems. CENELEC, Brussels, 2001.
[6] Bjørner D. Software Engineering, vol. 1: Abstraction and Modelling. Texts in Theoretical Computer Science. Springer, 2006.
[7] Bjørner D. Software Engineering, vol 2: Specification of Systems and Languages. Texts in Theoretical Computer Science. Springer, 2006.
[8] Bjørner D, George CW, Hansen BS, Laustrup H, Prehn S. A Railway System, Coordination'97, Case Study Workshop Example. Technical Report 93, Macau, 1997.
[9] Benveniste A, Berry G. The Synchronous Approach to Reactive and Real-Time Systems. Proc. of the IEEE, 1991, 79(9): 1270–282.
[10] Milner R, Tofte M, Harper R, MacQueen D. The Definition of Standard ML — Revised. MIT Press, 1997.
[11] Spillner A, Linz T, Schaefer H. Software Testing Foundations. dpunkt.verlag, Heidelberg, 2006.
[12] Eriksen LE, Pedersen B. Simulation of Relay Interlocking Systems. Technical Report IMM2007-05306. Lyngby: Technical University of Denmark, 2007.
[13] Haxthausen AE, Bliguet ML, Kjaer AA. Modelling and Verification of Relay Interlocking Systems. In 15th Monterey Workshop: Foundations of Computer Software, Future Trends and Techniques for Development, 2008.
[14] George C. RAISE Tools User Guide. Technical Report 227, Macau, February 2001. The tools are available free from http://www.iist.unu.edu
[15] Mosses PD, ed. CASL Reference Manual. Number 2960 in IFIP LNCS series. Springer, 2004.
[16] Spivey JM. The Z Notation: A Reference Manual. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
[17] Jones CB. Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, 2nd edition, 1990.
[18] Fitzgerald J, Larsen PG. Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998.
[19] Woodcock J, Davies J. Using Z–Specification, Refinement, and Proof. Series in Computer Science. Prentice Hall International, 1996.
[20] Owre S, Rushby JM, Shankar N. PVS: A Prototype Verification System. In Deepak Kapur, ed, 11th International Conference on Automated Deduction (CADE), volume 607 of Lecture Notes in Artificial Intelligence, Saratoga, NY, Springer-Verlag, 1992. 748–752.
[21] de Moura L, Owre S, Shankar N. The SAL Language Manual. Technical Report SRI-CSL-01-02, SRI International, 2003. http://sal.csl.sri.com

A RAISE Background

RAISE ("Rigorous Approach to Industrial Software") is a product consisting of a formal specification language RSL[2], an associated method[3], and a suite of tools[14].

A.1 Language

RSL is a formal, wide-spectrum specification language that enables the formulation of modular specifications which may be algebraic or model-oriented, applicative or imperative, and sequential or concurrent. A basic RSL specification is called a class expression and consists of declarations of types, values, variables, channels, and axioms. Specifications may also be built from other specifications by renaming declared entities, hiding declared entities, or adding more declarations. Moreover, specifications may be parameterized. Below we give a short survey of those kinds of declarations that are used in this paper. For a more detailed description of RSL, see Ref.[2].

A.1.1 Types and type declarations

User-Declared types may be introduced as abstract types, also called *sorts*, as known from algebraic specification languages like CASL[15], e.g.

type Colour

or may be constructed from built-in types and type constructors as in model-oriented languages like Z[16] and VDM[17,18], e.g.

type

Database = Key $\overrightarrow{m}$ **Nat-set**,

Key = **Text**

The built-in types include **Bool**, **Nat**, **Int**, **Real**, **Char**, and **Text** with the obvious meanings, and the type constructors include $\times$, **-set**, *, $\overrightarrow{m}$, $\rightarrow$, and $\xrightarrow{\sim}$ for constructing types of tuples, sets, lists, maps/tables (i.e. many-one relations), total functions, and partial functions, respectively. Types constructed in the second way are called *concrete* types. RSL also provides predicative subtypes, union and record types as known from VDM, and variant type definitions similar to data type definitions in ML[18].

With each type there is a collection of associated value constructors and operators. Most of these are the usual ones know from discrete mathematics, e.g. for set types, one of the operators is a union operator, $\cup$. Here we will only explain a few operators that are used in this paper and may be less obvious. For a list l, **elems** l gives the set of elements in the list. For a map m, that is a collection of associations of pairs of two values (a domain value and a range value), **dom** m and **rng** m give the set of domain values from the associations of m and the set of range values from the associations of m, respectively. For maps $m1$ and $m2$ having disjoint domains, $m1 \cup m2$ gives the

$m1$ and $m2$, $m1 \dagger m2$ gives the map that is the overriding of $m1$ with $m2$, i.e. the map that contains all associations of $m2$ and those associations of $m1$ that do not have domain values in common with $m2$. For example[″B1″ ↦5, ″B2″ ↦3] † [″B2″ ↦7, ″B3″ ↦4] = [″B1″ ↦5, ″B2″ ↦7, ″B3″ ↦4].

A.1.2 Values and value declarations

Values (i.e. constants and functions) can be specified in a signature[7]-axiom style as known from algebraic specification, e.g.

value

black, white : Colour

axiom

[distinct] black ≠ white

in a pre-post style, e.g.

value

square_root : **Real** → **Real**

square_root(x) **as** r **post** r ⩾ 0.0 ∧ r ∗ r = x

pre x ⩾ 0.0

or in an explicit style as known from model-oriented specification, e.g.

value

reverse : **Int*** → **Int***

reverse(l) ≡ **if** l = ⟨⟩ **then** ⟨⟩ **else** reverse(**tl** l) ^ ⟨ **hd** l⟩ **end**

A.2 Method

The RAISE method is based on *stepwise refinement*. The idea is to start with a high-level specification of the system and then in a number of steps to add details to the specification reflecting design decisions until a specification sufficiently concrete to be translated into a programming language has been achieved. For each step it should be proved that the new specification is a *refinement* of the specification of the previous step.

In RAISE *refinement* means theory extension. Intuitively this means that a specification $SP2$ is a refinement of a specification $SP1$, if:

- $SP2$ provides declarations of at least all those entities (types, values, variables and channels) that were declared in $SP1$ with the same names and types.
- All properties of $SP1$ must be logical consequences of the properties of $SP2$. The properties of a specification include the logical expressions in its axioms and equivalences that can be derived from its explicit value definitions. For example from the definition of the reverse function above one can derive the property:

$$\forall\, l : \mathbf{Int}^* \bullet \text{reverse}(l) \equiv \mathbf{if}\ l = \langle\rangle\ \mathbf{then}\ \langle\rangle\ \mathbf{else}\ \text{reverse}(\mathbf{tl}\ l)\ ^\frown\ \langle \mathbf{hd}\ l\rangle\ \mathbf{end}$$

In practise this means that one can for instance do the following in a refinement step:

- Axioms and declarations of new entities can be added.
- A sort declaration can be refined into a declaration of a concrete type, a variant type, a union type or a record type. Note that a concrete type can't be be refined into another concrete type as e.g. in VDM or Z[19].
- Axiomatic value definitions can be refined into explicit value definitions having the same signatures, provided that the axioms that are removed from the old specification can be proved to be logical consequences of the new explicit value definitions (and other properties of the new specification).
- An explicit value definition can be replaced by another explicit value definition if the two function bodies can be proved to be equivalent.
- Pre-Conditions of functions can be weakened, and post conditions can be strengthened.

A.3 Tools

With the method and language there is a suite of tools[14] supporting the construction, checking and verification of specifications and development relations, and for translating specifications in certain subsets of RSL into several other languages including SML[10], PVS[20], SAL[21] and C++.