Project: COMPASS

Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

A Dwarf Signal in CML

COMPASS White Paper WP04

September 2013

Public Document

`http://www.compass-research.eu`

Authors:
Simon Foster, Jim Woodcock, University of York, UK

Abstract:

This white paper presents an introduction to the COMPASS Modelling Language (CML) using a model of a Dwarf Signal. The paper introduces the states of the signal and the properties that must hold to ensure safety of the signal, and then introduces the types, functions and processes that model the state, safety properties and reactive behaviour in CML.

To demonstrate that such a system satisfies the contract imposed by its safety properties, the COMPASS theorem prover (based on Isabelle/HOL) can be used. Mechanisation of the example in the theorem prover is currently underway

A Dwarf Signal in CML

Simon Foster   Jim Woodcock

September 12, 2013

1 Introduction

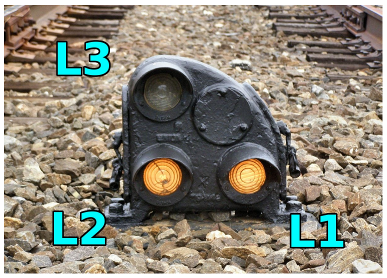

Figure 1: A Dwarf railway signal

A *Dwarf Signal* [6, 5] is a kind of railway signal which is used at the side of track when space is limited. An illustration is shown in Figure 1. It has three lamps which are displayed in different configurations to give instructions to train drivers. Clearly railway signals need to be safe and reliable in their implementation, and so in this paper we will give a precise specification to the Dwarf Signal, which will illustrate the use of the *COMPASS Modelling Language* [2] (CML). The signal's three lamps are named L1–L3, as illustrated, and different configurations of a signal can be written using set notation, for instance this signal is in $\{L1, L2\}$ which means **Stop**.

The signal has a total of four *proper states* which are the well-defined commands a signal can convey to a driver. These are enumerated in Figure 2. When all lamps are off (indicated by the empty set $\{\}$) the signal is in the **Dark** state, which is a power saving mode for when the signal is not in use. The three remaining proper states **Stop**, **Warning** and **Drive** are indicated by a combination of two lamps and correspond to the positions of an old-style semaphore signal.

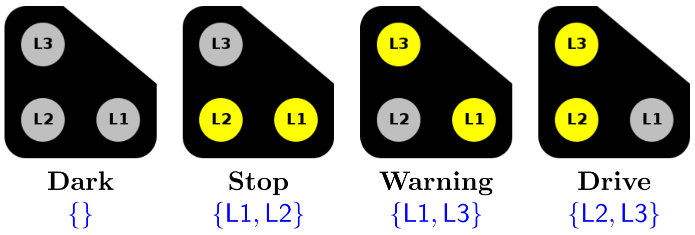

Figure 2: Dwarf Signal Proper States

Along with the four proper states there are also three *transient* states which describe the unstable states when a signal is moving from one proper state to another, since the signal may only light or extinguish one lamp at a time. These states are $\{L1\}$, $\{L2\}$ and $\{L3\}$. Finally there is the ambiguous state $\{L1, L2, L3\}$ which a signal should never display as it is meaningless. This makes a total of $2^3 = 8$ states.

To ensure the safety of this system, four safety properties are given:

1. Only one lamp may be changed at once
2. All three lamps must never be on concurrently
3. The signal must never be **Dark** except if the **Dark** aspect has to be shown or there is lamp failure
4. A change to or from **Dark** is allowed only from **Stop** or to **Stop**, respectively

These properties essentially form a *contract* with the signal, or the administrator of the signal, which ensures that the driver never observes an ambiguous state, and therefore never has to make an ill-informed decision. We can now proceed to specify the system in CML.

2 Dwarf Signal in CML

2.1 Signal State

We begin by specifying the basic types of our system.

```
types
  LampId      = <L1> | <L2> | <L3>
  Signal      = set of LampId
  ProperState = Signal
    inv ps == ps in set {dark, stop, warning, drive}

values
  dark: Signal    = {}
  stop: Signal    = {<L1>, <L2>}
  warning: Signal = {<L1>, <L3>}
  drive: Signal   = {<L2>, <L3>}
```

Table 1: Dwarf Signal basic types

The lamps are specified as an enumerated type `LampId`, and a `Signal` is then simply the set of lamps currently illuminated. A `ProperState` is a signal which is in one of the four proper states, indicated by four sets. With our basic types specified we can then specify the main state type of the Dwarf signal.

The signal state, `DwarfType`, is specified as a record with six fields. In order these refer to:

```
types
  DwarfType :: lastproperstate    : ProperState
               desiredproperstate : ProperState
               turnoff            : set of LampId
               turnon             : set of LampId
               laststate          : Signal
               currentstate       : Signal

  inv d ==
    (((d.currentstate \ d.turnoff) union d.turnon)
          = d.desiredproperstate)
    and
    (d.turnoff inter d.turnon = {})
```

Table 2: Dwarf Signal state specification

- the previous/current proper state the signal was in;
- the proper state we desire to reach;
- lamps we need to turn *off* to reach the desired proper state;
- lamps we need to turn *on* to reach the desired proper state;
- the actual (transient or proper) last state the signal was in;
- the actual current state the signal is in.

Along with fields we also specify an *invariant*, which impose further logic constraints on a CML type. In this case we use the invariant to ensure that only sane states are represented. There are two clauses to the invariant. The first ensures that the desired proper state is the current state, minus (\) the set of signals to turn off, plus (union) the set of signals to turn on. The second clause ensures there is no intersection of the lamps we wish to turn on, and those we wish to turn off. Clearly we can't simultaneously turn a lamp on and off.

2.2 Safety Properties

To ensure the safe functioning of the Dwarf Signal system we need to impose a number of safety properties. These properties should at all times be preserved by the system. In CML we specify them as a collection of five functions which are enumerated in Table 3.

NeverShowAll enforces that it should never be the case that all three lamps are on simultaneously. MaxOneLampChange requires that between any two states only one lamp can change from on to off, or off to on. ForbidStopToDrive enforces that the signal cannot transition straight from the stop state to the drive state – it must go via the warning state. DarkOnlyToStop and DarkOnlyFromStop together encode the requirement that a signal may only transition

```
functions
  NeverShowAll: DwarfType -> bool
  NeverShowAll(d) == d.currentstate <> {<L1>,<L2>,<L3>}

  MaxOneLampChange: DwarfType -> bool
  MaxOneLampChange(d) ==
    card ((d.currentstate \ d.laststate)
         union (d.laststate \ d.currentstate)) <= 1

  ForbidStopToDrive : DwarfType -> bool
  ForbidStopToDrive(d) ==
    (d.lastproperstate = stop
     => d.desiredproperstate <> drive)

  DarkOnlyToStop : DwarfType -> bool
  DarkOnlyToStop(d) ==
    (d.lastproperstate = dark
    => d.desiredproperstate in set {dark,stop})

  DarkOnlyFromStop: DwarfType -> bool
  DarkOnlyFromStop(d) ==
    (d.desiredproperstate = dark
    => d.lastproperstate in set {dark,stop})
```

Table 3: Dwarf Signal: Safety Properties

from dark to stop, and to dark from stop – a signal in warning or drive should not become stop directly. With our collection of safety properties which can describe the safe version of the Dwarf Signal state:

```
types
  DwarfSignal = DwarfType
  inv d == NeverShowAll(d) and
           MaxOneLampChange(d) and
           ForbidStopToDrive(d) and
           DarkOnlyToStop(d) and
           DarkOnlyFromStop(d)
```

2.3 Reactive Behaviour

Syntax	Description
Stop	Deadlocked process
Skip	Null behaviour
`a -> P`	Communicate on `a` then behave like `P`
`a?v -> P`	Input value `v` over channel `a` then do `P`
`a!v -> P`	Output value `v` on channel `a` then do `P`
`P ; Q`	Execute action `P` followed by `Q`
`P [] Q`	Pick `P` or `Q` based on the first communication
`P [|{a,b,c}|] Q`	Execute `P` and `Q` in parallel, with synchronisation allowed on `a`, `b` and `c`
`[cond] & P`	allow execution of `P` only if `cond` holds

Table 4: CML process combinator selection

The Dwarf Signal is a *reactive* system; it waits for stimuli and behaves accordingly. To specify these sorts of aspects of a system we need to use a suitable formalism. In CML we support the specification of *CSP* processes. CSP (Communicating Sequential Processes) is a *process calculus* which specifies behaviour in terms of concurrent processes which communicate on *channels*. A channel is a two-ended medium with a single listener and a single speaker. A channel can therefore be used to send information between one processes and another. A process in CML consists of five parts:

- *channels* to communicate on, optionally carrying data;
- *state variables* to read from and write to;
- *operations* acting on the state, with pre/postconditions;
- *actions* which describe reactive behaviours;

- *process body*, the main behaviour of the process.

Unlike CSP processes, CML processes are *stateful*, though there is no state sharing between the individual constituent actions. Only channels may be used to share information. CML supports a variety of *process combinators* to specify reactive behaviour, many of which are borrowed from CSP (see Table 4).

As an example we could specify the following simple process:

```
channels
  a: int
  b: int

process Simple = begin

actions
  ACT1 = a?v -> b!(v * 2) -> Skip
  ACT2 = a!5 -> Skip
@

ACT1 [|a|] ACT2

end
```

Table 5: A basic CML process

This specifies two channels, `a` and `b`, both of which carry an integer. The process `Simple` consists of two constituent actions. `ACT1` waits for an input on `a`, and then communicates this value doubled on `b`, and finally terminates, indicated by `Skip`. `ACT2` sends 5 on `a` and then terminates. The main action of the processes is simply the two actions in parallel, with shared channel `a`. So if run the result will be an output of 10 on `b`.

We proceed to specify the Dwarf Signal process.

There are a total of 5 channels, `init` which instructs the system to initialise, `light` and `extinguish` on which the signal can be instructed to light or extinguish a given lamp, `setPS` to instruct the signal to begin transitioning to a given proper state, and `shine` which the process uses to display the currently lit lamps. The state of the process consists of a single variable `dw` which is an instance of `DwarfType`, the state record from Table 2. Next we specify some operations to act on this state.

```
operations
  Init : () ==> ()
  Init() ==
    dw := mk_DwarfType(stop, {}, {}, stop, stop, stop)
```

First we specify the `Init` operation, which simply sets up the initial value for state variable `dw`. Its type is $() \Longrightarrow ()$, meaning it takes no inputs and produce

```
channels
  init
  light: LampId
  extinguish: LampId
  setPS: ProperState
  shine: Signal

process Dwarf = begin

state
  dw : DwarfSignal
```

Table 6: Dwarf Signal: Channels and State

no outputs, performing just a state update. The initial value is constructed using the record constructor `mk_DwarfType`. It has the current state, proper state and next state as **Stop** and the sets of lamps to turn on and off both as empty {}. It is therefore a completely stable state, and is awaiting instructions.

```
SetNewProperState: (ProperState) ==> ()
SetNewProperState(st) ==
  dw := mk_DwarfType( dw.currentstate
                    , dw.currentstate \ st
                    , st \ dw.currentstate
                    , dw.laststate
                    , dw.currentstate
                    , st)

  pre dw.currentstate = dw.desiredproperstate and
      st <> dw.currentstate
```

This next operation takes a new proper state and updates the state record to indicate this. It also calculates the set of lamps which must be lit and extinguished for this state to be reached, and adds these to the state. This operation has a precondition that the current state must be the desired proper state – i.e. the system must have stabilised. It also requires that the new proper state be different from the one we are currently in.

```
TurnOn: (LampId) ==> ()
TurnOn(l) ==
  dw := mk_DwarfType( dw.lastproperstate
                    , dw.turnoff \ {l}
                    , dw.turnon \ {l}
                    , dw.currentstate
                    , dw.currentstate union {l}
```

```
                   , dw.desiredproperstate)

  pre l in set dw.turnon

TurnOff : (LampId) ==> ()
TurnOff(l) ==
  dw := mk_DwarfType( dw.lastproperstate
                    , dw.turnoff \ {l}
                    , dw.turnon \ {l}
                    , dw.currentstate
                    , dw.currentstate \ {l}
                    , dw.desiredproperstate)

  pre l in set dw.turnoff
```

TurnOn and TurnOff add and remove, respectively, a lamp from the set of lit lamps. The both require that the lamp to be turned on or off is in the set turnon or turnoff respectively. These values are then also updated accordingly, as the same lamp cannot be lit or extinguished twice.

With manipulations for the state specified, we can now specify the reactive behaviour of the system.

```
actions
  DWARF =
    (  (light?l -> TurnOn(l); DWARF)
    [] (extinguish?l -> TurnOff(l); DWARF)
    [] (setPS?l -> SetNewProperState(l); DWARF)
    [] shine!dw.currentstate -> DWARF)

@ init -> Init() ; DWARF
```

Table 7: Dwarf Signal Reactive Behaviour

The single action, DWARF, consists of an event loop which make a choice between four sequences. If a communication is received on light, then the operation TurnOn is called on the given lamp and then the action recurses to its initial behaviour. If a communication is received on extinguish then the TurnOff operation is called. If setPS receives a new proper state, then the new proper state is set. Finally, the signal is always capable of communicating on shine the current lamp status, which gives us a way of observing it.

The main action waits for an instruction to init, and when received it calls the Init operation and then enters the main event loop.

We can also describe some test actions which exercise the behaviour of this process, as in Figure 8. These test are all composed with the main DWARF action and explore some its different behaviours.

TEST1 is a valid behaviour of the initialised system, where we transition from stop to drive via warning, lighting and extinguishing the appropriate lamps along the way. TEST2 starts the same as TEST1, but flips the order in which L2 and L3 are extinguished and lit, respectively. This violates the safety property NeverShowAll, and if simulated within the COMPASS tool [3] an error will be raised in the second transition. TEST3 begins by transitioning from stop to dark (a valid move), but then immediately tries to go to warning. This violates the safety property DarkOnlyToStop, and again the COMPASS tool will complain. Finally TEST4 tries to go straight from stop to drive, violating ForbidStopToDrive.

This test suite, though by no means exhaustive, demonstrates how we can exercise the Dwarf Signal to ensure the safety of the system. In the future it will be possible to use the COMPASS theorem prover [4] to prove that the invariants can never be violated.

```
-- Working test
TEST1 = setPS!warning -> extinguish!<L2> -> light!<L3>
      -> setPS!drive -> extinguish!<L1> -> light!<L2> -> Stop

-- Try to turn on 3 lights simultaneously
TEST2 = setPS!warning -> light!<L3> -> extinguish!<L2>
      -> setPS!drive -> extinguish!<L1> -> light!<L2> -> Stop

-- Try to go from dark to warning
TEST3 = setPS!dark -> extinguish!<L1> -> extinguish!<L2>
     -> setPS!warning -> light!<L1> -> light!<L2> -> Stop

-- Try to go from stop to drive
TEST4 = setPS!drive -> extinguish!<L1> -> light!<L3> -> Stop

DWARF_TEST1 = DWARF [|setPS,light,extinguish|] TEST1
DWARF_TEST2 = DWARF [|setPS,light,extinguish|] TEST2
DWARF_TEST3 = DWARF [|setPS,light,extinguish|] TEST3
DWARF_TEST4 = DWARF [|setPS,light,extinguish|] TEST4
```

Table 8: Dwarf Signal Tests

3 Conclusion

We have demonstrated the use of CML in specifying a real-life system, namely the *Dwarf Signal*, in terms of its state, safety properties and reactive behaviour. The COMPASS tools can be used in such a system to demonstrate its safety by ensuring it satisfies its contract. This example is currently being mechanised in the COMPASS theorem prover, which is based on *Isabelle/HOL* [1]. This will enable mechanical verification of the systems contractual obligations.

References

[1] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *8th International Symposium on Frontiers of Combining Systems (FroCoS 2011)*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.

[2] Jeremy Bryans, Samuel Canham, Ana Cavalcanti, Andy Galloway, Thiago Santos, Augusto Sampaio, and Jim Woodcock. CML Definition 2. Technical report, COMPASS Deliverable, D23.3, March 2013. Available at http://www.compass-research.eu/.

[3] Joey W. Coleman, Anders Kaels Malmos, Rasmus Lauritsen, and Luís D. Couto. Second release of the COMPASS tool — user manual. Technical report, COMPASS Deliverable, D31.2a, January 2013.

[4] Simon Foster and Richard J. Payne. Theorem proving support - user manual. Technical report, COMPASS Deliverable, D33.2a, September 2013.

[5] A. A. McEwan and J. C. P. Woodcock. A refinement based approach to calculating a fault-tolerant railway signal device. In René Jacquart, editor, *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions*, volume 156 of *IFIP Advances in Information and Communication Technology*, pages 621–627. Springer, 2004.

[6] J. C. P. Woodcock. Montigel's Dwarf, a treatment of the dwarf-signal problem using CSP/FDR. In *Proc. 5th FMERail Workshop*, September 1999.