Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS Interest Group Workshop

Deliverable Number: D51.4

Version: 1.0

Date: July 2013

Public Document

http://www.compass-research.eu

Contributors:

Steve Riddle, Newcastle
Samuel Canham, York
Simon Foster, York
Jim Woodcock, York

Editors:

Steve Riddle, Newcastle

Reviewers:

Andy Galloway, York
Jon Warwick, Newcastle
Augusto Sampaio, UFPE

Document History

Ver	Date	Author	Description
0.1	24-06-2013	Steve Riddle	Draft in preparation
0.2	10-07-2013	Steve Riddle	Final draft for review
1.0	31-07-2013	Steve Riddle	Issue 1

Abstract

This deliverable provides a short summary of the first COMPASS Interest Group workshop, which took place in Bristol, UK on 22 May 2013. The deliverable includes a summary of presentations, points raised in discussion and outlines future plans for the development of the Interest Group.

The workshop consisted of five talks which introduced the motivation for COMPASS, illustrated the applicability of the COMPASS technologies in a System of Systems domain, presented the guidelines for requirements and architecture description, summarised the CML language and gave a demonstration of model-based testing. This was followed by discussion of issues raised. A further workshop will be held to show the COMPASS approach applied to case studies.

Appendices to this report present a list of attendees and a summary introduction to CML derived from the presentation given at the workshop. This CML introduction is included in response to a recommendation from the project reviewers to present CML to the COMPASS Interest Group.

Contents

1 Introduction

This deliverable summarises the first COMPASS Interest Group workshop, which took place in Bristol, UK on 22 May 2013. The workshop aimed to present the motivation for, and current status of the COMPASS methods and tools, and give an opportunity for members of the COMPASS Interest Group to discuss issues relating to Systems of Systems, their own experience and reactions to the presentations of COMPASS methods and tools.

Currently there are 15 members of the COMPASS Interest Group, including six from the United Kingdom, and another six from mainland Europe. With this in mind, it was felt appropriate to hold the first CIG workshop in the UK, and Bristol was chosen as a convenient location. Representatives of five companies accepted the invitation to attend the workshop; these were Roke Manor, Rolls Royce, Jaguar Land Rover, Altran and BAE SYSTEMS.

The discussion was facilitated with the use of sticky (Post-it) notes issued to attendees before the presentations. Attendees were encouraged to record impressions, questions and suggestions as they occurred to them during the talks, and a selection of notes was then used to structure the final discussion.

The document is structured as follows. Section 2 gives an overview of the content of the talks presented at the workshop. Section 3 summarises discussion points and Section 4 presents plans for the future. Section 5 concludes the report. Appendices present the list of attendees (Appendix A) and a summary of the presentation on the COMPASS Modelling Language (CML) which presents a small illustrative case study (Appendix B).

2 Workshop content

The following is a summary of the talks presented at the Workshop.

- Welcome and Introductions – John Fitzgerald, Newcastle University

  This talk presented a general introduction and summary of COMPASS, including motivating examples (Traffic Management, Home Audio-Visual Automation, Energy Management), the project objectives and the model-based approaches, languages, tools and analysis techniques being developed. This was followed by a brief example of the COMPASS approach using the Accident Response case study, and a forward look.

- Systems of Systems Challenges and the COMPASS tools – Klaus Kristensen, Bang & Olufsen

  The talk presented the COMPASS approach from the point of view of a technology provider, to show how the results of COMPASS are expected to help them achieve their goals of managing SoS requirements, deal with emergent behaviour, improve software quality and SoS robustness, and enable long-term streamlining of SoS development by bringing together independent software teams. This was illustrated using applications of architectural modelling, algorithm modelling, testing and requirements engineering in the domain of distributed Audio-Visual streaming devices.

- Model-based Systems Engineering for Systems of Systems – Simon Perry, Atego

  Presenting the contribution of Atego in COMPASS, this talk summarised the guidelines for SoS development. This included the approach to requirements, architecture, integration and system engineering. The ontology, framework and processes for SoS requirements engineering and architecture were presented with SysML models, and there was further discussion on patterns and tool support.

- Introducing the COMPASS Modelling Language – Samuel Canham, University of York

  The COMPASS Modelling Language (CML) was introduced using a simple case study, the Dwarf Railway Signal example. This example was used to illustrate the key features of a CML model: types, functions, operations and processes. Appendix B summarises the CML language based on the content of this talk. This appendix is included in order to present the CML language to CIG members, as recommended by the project reviewers.

- Advanced Model-Based Testing With RT-Tester and Artisan Studio – Jan Peleska, University of Bremen

  The final presentation summarised the goals of Model-Based Testing and gave a demonstration of RT-Tester and its integration with Artisan Studio. This was followed by a summary of work in progress: the development of specialised test strategies for SoS, and an integrated V&V workflow between the COMPASS tools, Artisan Studio and RT-Tester.

The final presentation was followed by a discussion session, which is summarised in the next section.

3 Discussion

As noted above, discussion was facilitated by the use of sticky notes which were issued to attendees before the presentations. This did not preclude discussion during the workshop itself, but there was an opportunity for more focused discussion following the presentations, using a selection of the notes.

Notes and queries arising from this process can be divided into 6 broad categories. These categories are summarised below.

1. Abstraction, Constraints and Process Issues

   There is a trade-off to be made between **abstraction** and **fidelity**. This is a standard question with abstraction: what information to leave out is a matter of engineering judgment.

   We present an example of CML based on **bottom-up** definition of behaviour of constituent systems. How can we deal with SoS constraints, and provide analysis **top-down** as well? Can we flow down SoS level constraints on to constituent systems?

2. Emergence, Properties, Fault handling

   Robustness is a desirable property of a SoS. Dealing with exceptions is crucial - when specified or assumed conditions aren't met, what is the behaviour? Emergent behaviour is often associated with transient conditions, such as initialisation or shutdown.

   How to define, combine, trade non-functional properties such as security, availability, safety, performance.

3. Viability of the approach

   How applicable is the approach to different development life-cycles, for example an iterative approach - and how tool agnostic is it?

   CML script is not particularly readable to a novice to the language - is this sensible in a systems engineering context with so many stakeholders?

How mature must a model be before a “sensible” level of analysis and CML translation can be performed?

4. SoS vs System

   A general concern was expressed that the project is addressing Systems, rather than Systems of Systems. What is *uniquely* SoS oriented in the solutions being investigated? This concern is partly due to the nature of some of the examples presented, as for the purposes of explanation we need to have examples that are simplified to put across the language details.

5. Terminology

   Some concern expressed about terminology, for example “Ontology” versus “Domain Specific Language” and the phrase “use case”, as its meaning is not always the same as the standard UML/SysML term known throughout industry.

6. Collaboration

   What collaboration exists between this and other EU projects, and other initiatives. We have close links with DANSE, ROAD2SOS and T-AREA-SOS among other EU projects, and the INCOSE Working Group in Systems of Systems. Other links and initiatives include the OMG and the related ALF (UML Action Language).

The summary of issues presented above gives a flavour of some of the discussion during the workshop. To encourage further debate which can include other members of the CIG and the project as a whole, a discussion group is being set up on LinkedIn. Other plans for the future are discussed in the following section.

4 Plans for the future

It was agreed that further CIG workshops should be held, to include a second version of this initial workshop (located in central Europe) and a follow-on workshop centred on the case studies. Consideration is currently being given to the timing and location. We will also look to the Summer School which is to be coordinated in the final phase of the project in summer 2014, which will include tutorials given by research and industrial partners to encourage take-up of the projects scientific findings subsequent to the project.

5 Conclusion

The workshop has been viewed as an unqualified success in presenting the current status of the project and the upcoming method guidelines and tool releases. The discussion showed there is great deal of interest in the COMPASS technology within the CIG. It also served to highlight potential pitfalls, e.g. with respect to the scalability and viability of the approach, providing a valuable source of experience from which to influence the development of the method.

In the short-term, the aspiration is for a larger CIG having greater interaction with the project. Activities to achieve this include a twitter feed (`@COMPASS_SoS`) which was introduced to CIG members at the workshop, and use of an accessible forum for ongoing discussion, as outlined above.

A List of attendees

CIG member attendees:

- Ken Richardson, Roke Manor Research Ltd
- Dave Banham, Rolls Royce
- Ross McMurran, Jaguar Land Rover
- Ian Gallagher, Altran UK
- Kevin Dockerill, BAE SYSTEMS

Project attendees:

- Zoe Andrews, John Fitzgerald, Jon Warwick, Jodi Hossbach, Steve Riddle (Newcastle University)
- Samuel Canham (University of York)
- Simon Perry (Atego)
- Jan Peleska (University of Bremen)
- Klaus Kristensen (Bang & Olufsen)
- Margherita Forcolin (Insiel)

B CML Summary

This appendix presents a summary of the COMPASS Modelling Language (CML), summarised from the presentation and notes written by Samuel Canham, Simon Foster and Jim Woodcock and presented by Samuel Canham at the CIG workshop, 22 May 2013.

CML is:

- a formal language for specifying Systems of Systems
- draws input from formal languages *VDM* and *Circus*

A CML document consists of

- **types** with invariants, e.g.
  - basic types: **bool**, **int**, **string**, **real** etc.
  - enumerations ("quote" type)
  - sets
  - maps
  - records
- **functions** with pre and postconditions
- **operations** which act on a *state*, which represents persistent data
- **processes** to represent *concurrent* and *reactive* behaviour

B.1 Dwarf Signal Example

These language features are illustrated by an example, the Dwarf Railway Signal. Used where space is limited, Dwarf signals consist of three lamps each of which may be lit or unlit. Their interpretation is based on the old semaphore signaling system in which a bar (arm) was raised and lowered into vertical, horizontal and diagonal positions.

The three lamps are named L1-L3. There are four "proper states" a signal may be in: Dark, Stop, Warning and Drive. These are indicated by the signal configurations in Figure 1. As well as these states there are 4 "transient states" which the signal can be in when moving from one state to another.

These have no meaning, but the driver should interpret a transient state in a "safe" manner – e.g. not drive when stop could be the state.

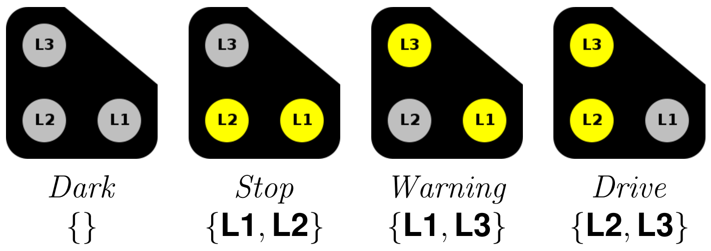
L3
L2
L1
Dark
$\{\}$
Stop
$\{\textsf{L1}, \textsf{L2}\}$
Warning
$\{\textsf{L1}, \textsf{L3}\}$
Drive
$\{\textsf{L2}, \textsf{L3}\}$

Figure 1: Proper States for the Dwarf Signal

The transient states are $\{\textsf{L1}\}, \{\textsf{L2}\}, \{\textsf{L3}\}, \{\textsf{L1}, \textsf{L2}, \textsf{L3}\}$

B.2 Safety Requirements

The signal has several *safety requirements* which must be obeyed by the signal software for the signal to be considered safe:

1. Only one lamp may be changed at once
2. All three lamps must never be on concurrently
3. The signal may not go straight from *Stop* to *Drive*
4. The change to and from *Dark* is allowed only from *Stop* and to *Stop*

A typical trace for the lamp would be the transition from *Dark* to *Drive*, which is achieved one lamp at a time as shown in Figure 2.

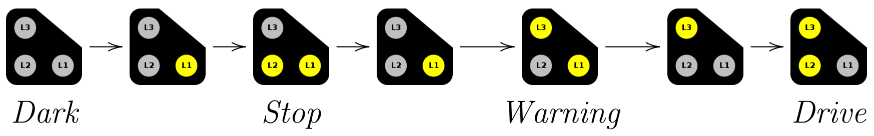

Figure 2: Typical trace for the signal: Moving from Dark to Drive

B.3 Basic types

The basic types in CML for the Dwarf Signal are as follows.

```
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>}
```

Here, `LampId` is an enumerated type representing one of the three lamp identifiers; `Signal` is a set of lamp identifiers and represents the possible states of the signal. `ProperState` is a subtype of Signal with an invariant requiring that the signal be in one of the proper states, as indicated by the values below.

B.4 Dwarf Signal State

The Dwarf Signal State captures the persistent data for the Dwarf Signal: this includes the last `ProperState` the Signal was in, the set of lamp identifiers representing the lamps that must be turned off in order to reach the desired state, the lamps that must be turned on to reach the desired state, the last and current (possibly transient) states that the lamp is now in, and finally the target (desired) `ProperState` we wish to change to next.

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

B.5 Dwarf Signal State - Invariants

As well as the basic record definition, we also need to add two *invariants* which will ensure internal consistency of the signal state.

- The first clause (over the page) ensures that the desired state (i.e. the state we wish to go to next) is reached by turning the lamps in `turnoff` off, and those in `turnon` on.
- The second requires that we can't simultaneously require to turn a lamp on and off (the intersection of `d.turnoff` and `d.turnon` is empty)

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

B.6 CML Processes

A process in CML represents concurrent and reactive behaviour. A process specification will include:

- *channels* to communicate on, optionally carrying data
- *state variables* to read and write to
- *operations* acting on the state
- *actions* which describe reactive behaviours
- *process body*, the main behaviour of the process

B.6.1 CML process syntax

The table below summarises the syntax for a basic CML process.

Syntax	Description
`Stop`	Deadlocked process
`Skip`	The correctly terminating process
`a -> P`	Communicate on `a` then behave like `P`
`a?v -> P`	Input value to variable `v` over channel `a` then do `P`
`a!e -> P`	Output value `e` on channel `a` then do `P`
`P ; Q`	Execute process `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 required on `a`, `b` and `c`
`[cond] & P`	allow execution of `P` only if `cond` holds

B.6.2 A basic CML process

The example below has two channels, `a` and `b`, and a main process called `Simple`. We write an `@` symbol to indicate the main action of the process, and then write down process syntax.

The main action consists of the parallel composition of two actions which can communicate only on the shared channel `a`. The left-hand action inputs a value on channel `a` which it binds to the variable `v`. It then sends on `b` the value `v` multiplied by 2, and then terminates activity with Skip. The right-hand action simply sends 5 on `a`.

```
channels
  a: int
  b: int

process Simple = begin
@
(a?v -> b!(v * 2) -> Skip) [|a|] (a!5 -> Skip)
end
```

This results in the following pattern of behaviour:

$$\begin{tikzcd}
\texttt{(a?v -> b!(v * 2) -> Skip) [|a|] (a!5 -> Skip)} \arrow[d, "\texttt{a.5}"] \\
\texttt{(b!(v * 2) -> Skip) [|a|] (Skip)} \arrow[d, "\texttt{b.10}"] \\
\texttt{(Skip) [|a|] (Skip)}
\end{tikzcd}$$

Here a 5 is first sent on `a`, and then a 10 on `b`.

We can now describe the overall Dwarf Signal process.

B.7 Dwarf Operations and Processes

We have 5 channels:

- `init`, which tells the Dwarf Signal to initialise
- `light`, on which the signal may be commanded to turn on a lamp
- `extinguish`, on which the signal may be commanded to turn off a lamp
- `setPS`, on which the signal may be commanded to begin transitioning to a new proper state
- `shine`, to represent our ability to observe the state of the lamp

The process also has a state variable, using `DwarfType` which we created before.

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

process Dwarf = begin

state
  dw : DwarfType
end
```

B.7.1 Init operation

The first operation is the `Init` operation which simply places the signal in the Stop proper state. We specify this by means of an assignment command and a post condition, which gives the state of the signal after the operation was executed.

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

    post dw.lastproperstate = stop and
         dw.turnoff = {} and
         dw.turnon = {} and
         dw.laststate = stop and
         dw.currentstate = stop and
         dw.desiredproperstate = stop
```

B.7.2 Set New Proper State

This operation sets a new proper state for the signal to transition into. As a precondition we require that the the signal has already stabilised into its proper state – it must not be in a transient state – which we ensure by requiring that the current state be the desired proper state. We also require that the new desired proper state must be different from the old one – the command must serve some purpose. Internally, the assignment sets up the lamps to be turned on and those to be turned off by means of the set different operator \.

```
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
```

B.7.3 Turn On / Turn Off

The `TurnOn` operation turns a signal on. It requires that the signal must desire to turn the lamp on, by its presence in the set `turnon`. It adds the lamp to the set of lights that are on (`currentstate`), and removes it from the lamps to be turned on (`turnon`).

`TurnOff` is defined similarly.

```
TurnOn: (LampId) ==> ()
TurnOn(l) ==
  dw := mk_DwarfType( dw.lastproperstate
                    , dw.turnoff \ {}
                    , 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 \ {}
                    , dw.currentstate
                    , dw.currentstate \ {l}
                    , dw.desiredproperstate)
  pre l in set dw.turnoff
```

B.7.4 Dwarf Signal Process

With our operations defined we are in a position to create an agent to describe the reactive behaviour of our signal. We create an agent `DWARF` which makes a choice of one of four things:

- If we receive the command to light, we turn on the said lamp
- If we receive the command to extinguish, we turn off the said lamp
- If we receive the command to setPS, we set the new proper state to transition to
- The signal can always shine out its current state

The main action of our process initialises the signal when told to on init, and then starts the core behaviour described in `DWARF`. The specification of the action given here is simplified for the purpose of introducing the notation; a full specification would need to ensure that the preconditions of the operations `TurnOn`, `TurnOff` etc are satisfied, so that we can guarantee that the invariant on the state variable holds.

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

@

init -> Init() ; DWARF
```

B.7.5 A bad trace

Not all traces have good results. Figure 3 illustrates a bad trace.

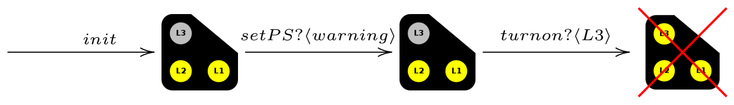

Figure 3: A bad trace, violating the safety property that all three lamps are never on at the same time

This violates the safety property "All three lamps must never be on concurrently", requirement 2 in Section B.2. We need to consider how to represent these properties.

B.8 Adding Safety Properties

We can represent safety properties using boolean functions on the state: for example the safety property "All three lamps must never be on concurrently" can be represented as the function `NeverShowAll`:

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

Safety property 1, "Only one lamp at a time may change" can be represented as:

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

Safety property 3, "The signal may not go straight from *stop* to *drive*" is represented as:

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

Safety Property 4, "The change to and from *dark* is allowed only from *stop* and to *stop*" can be separated into two properties: first, "The only proper state allowed to follow *dark* is *stop*":

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

Second, "The only proper state allowed to precede *dark* is *stop*":

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

These safety properties are now all added to the invariant for the `DwarfSignal`.

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

Following the establishing of this invariant, we can define further actions to modify the Dwarf state and show, using the COMPASS tools, whether the invariant is respected. The example is dealt with in more detail in a forthcoming COMPASS white paper due to be published in August 2013.

B.9 Conclusion

This appendix has introduced features of CML through a simple example. More details of the language are presented in the following deliverables:

- Formal mathematical basis in UTP (**D23.2**)
- Editor, type-checker and basic simulator (**D31.2**)
- Linking between CML and SysML (**D22.4**)

Work in progress includes:

- Mechanical verification of the semantics
- Model-checking (**D33.1**)
- Theorem prover plugin for COMPASS tool (**D33.2**)
- Full simulation support (**D32.2**)
- Refinement of CML specifications (**D33.4**)