Features of CML: a formal modelling language for Systems of Systems

J. Woodcock*, A. Cavalcanti*, J. Fitzgerald†, P. Larsen‡, A. Miyazawa*, and S. Perry§

*University of York, United Kingdom, {jim.woodcock, Ana.Cavalcanti, alvaro.miyazawa}@york.ac.uk
†Newcastle University, United Kingdom, john.fitzgerald@newcastle.ac.uk
‡Aarhus University, Denmark, pgl@iha.dk
§Atego, United Kingdom, simon.perry@atego.com

Abstract — We discuss the initial design for CML, the first formal language specifically designed for modelling and analysing Systems of Systems (SoSs). It is presented through the use of an example: an SoS of independent telephone exchanges. Its overall behaviour is first specified as a communicating process: a centralised telephone exchange. This description is then refined into a network of telephone exchanges, each handling a partition of the set of subscribers (telephone users). The refinement is motivated by a non-functional requirement to minimise the cabling required to connect geographically distributed subscribers, who are clustered. The exchanges remain as independent systems with respect to their local subscribers, whose service is unaffected by the loss of remote exchanges.

Keywords: architecture, SysML, modelling, specification, refinement, evolution, formal analysis, VDM, *Circus*, CML, semantics, UTP, enslavement pattern.

1. Introduction

The design of products and services that exploit Systems-of-Systems (SoS) technology is in its infancy. It is hampered by the complexity caused by the heterogeneity and independence of SoS constituent systems. State-of-the-art SoS engineering lacks models and tools to help developers make trade-off decisions during design and evolution, and to assist in working out and recording precise contracts between constituents and the global SoS. This leads to sub-optimal design and expensive rework during integration and in service.

COMPASS (Comprehensive Modelling for Advanced Systems of Systems)[1] is augmenting existing industry tools and practice with a modelling language in which SoS architectures and contracts can be expressed. A formal semantic foundation—the first to be developed specifically for SoS engineering—enables analysis of global properties. The language and methods will be supported by an open tools platform [1] with prototype plug-ins for model construction, dynamic analysis by simulation and test automation, static analysis by model checking and proof, and links to an established architectural modelling language, SysML [2]. These strengthened foundations and tools will support enhanced methods that help users embed this new technology in industrial practice.

Our approach is based on CML (COMPASS Modelling Language) with formal semantics, and methods and tools that take advantage of this. It uses an integration of the Systems Modelling Language, SysML, and CML, where developers can start if they wish from a graphical architectural view that is readily communicated to stakeholders.

CML is founded on the well-established *Circus* [3] (which in turn is based on Z [4] and CSP [5]) and VDM [6] formalisms, and includes SoS-specific aspects, such as contracts for constituent systems. A contract is a specification that describes the assumptions and guarantees of a CML model, and whose compliance is checked using refinement. A CML model is a collection of process definitions; each process encapsulates a state and operations written in VDM and interacts with the environment via synchronous communications, like in *Circus*. Using CML, many different kinds of analyses can be conducted, and some will be presentable at the SysML level. The semantics of CML is currently being developed using UTP [7].

In Section 2, we describe the approach to modelling in CML; in Section 3, we describe the

[1]COMPASS is a EU FP7 project. The COMPASS website is www.compass-research.eu.

CML model of our telephone exchange SoS; and in Section 4, we draw some conclusions from the work.

2. Modelling in CML

A central notion is that of conformance between models: one model C is correct with respect to another model A, if every behaviour of C is also a behaviour of A. When conformance holds, then C must inevitably pass every test carried out with respect to A. This notion of behavioural refinement offers a way of defining contracts: we would like A to be a simple presentation of all acceptable behaviours (the contract); C can then be more ingenious about how to implement the contract. A formal notion of refinement allows us to check the correctness of C against the contract defined by A.

CML can be used to model existing systems, compose them into an SoS, define suitable contracts for this composition, and check that the contracts are fulfilled. In this model-based development, an SoS specification can be traceably decomposed into an architecture and a collection of requirements on constituent systems expressed as contracts. The constituent systems can then be further decomposed or implemented by procurement of specific systems. Other requirements can be shown to emerge as a consequence of executing constituent systems.

The overall contract must describe the behavioural properties of the SoS, as well as specific policies and constraints for coordinating constituent systems and their workflow. It will provide global invariants that may be inexpressible at a lower level, and so can be used to constrain emergent behaviour. As an example, consider an SoS that manages the clearing system for a group of banks. A global invariant would reconcile the amount of money coming into the clearing system, the money moving between banks, and the money leaving the clearing system. This is a clear and intuitive invariant of the global view of the system, but it is obviously not an invariant of any individual bank.

The architectural description needs to represent the topology of the SoS and the interactions between the different constituent systems. These channels may require additional properties of bandwidth, delays, and potential faults, as well as the more obvious behavioural protocols for correct operation. Some SoSs will be dynamically evolving, and CML will provide mechanisms to describe mobile channels and mobile processes with inspiration from [8].

3. Example: Telephony System

We describe a small but realistic example using CML. We restrict ourselves to a high-level informal overview of the models involved, omitting formal details and the use of SysML to document the architecture at key phases. Although the development described results in a homogeneous SoS, the pattern is equally applicable to heterogeneous constituents.

A. Abstract Model

We start with a model of the functionality of an automated telephone exchange for connecting subscribers' calls. We assume that a caller identifies the recipient when initiating a call in a single, atomic action. This simplifies our presentation; there is no conceptual difficulty with separating this into seizing a line and supplying the recipient's identity, and even decomposing this into dialling a number digit by digit. Only the caller can clear a telephone call: if a recipient tries to clear a call, then it becomes merely suspended; if the recipient lifts the receiver again, then the call is re-established.

The exchange is specified as a single CML process, encapsulating a state described in VDM, recording the status of every subscriber and every call currently in progress. The operations on this state are made reactive with CML's CSP notation, linking events to the effect on the state. The model gives us a contract for the service provided by the exchange, and we can augment this model with guarantees about quality of service, such as the time to connect a call to the engaged or ringing tone.

B. System of Systems

Subscribers are geographically distributed, but clustered. The initial model requires extensive cabling, and a non-functional requirement is to reduce cabling costs. One way to do this is to install a separate exchange at each cluster, and provide trunk cabling between these exchanges. A subscriber then makes a call to the nearest exchange; the call is either serviced locally or routed to a remote exchange, as appropriate. This suggests an SoS architecture embedding instances of the simple exchange in a suitable topology, with the addition of trunk signalling between exchanges. This SoS can then be shown to refine the original simple exchange. This demonstrates that the required service is being delivered correctly, in spite of a more elaborate implementation. Crucially, the subscriber need not be aware of the way that the service is implemented,

either as a single exchange, or as a collection of exchanges with trunk signalling between them.

An emergent property of the new architecture is its fault tolerance. It may be that a single fault in an exchange could cause the exchange's entire service to be lost; but in the new architecture, the only loss would be the telephone calls to and from subscribers local to that exchange. This capability also supports evolution of the system in a way that the single, centralised model does not. An individual exchange can be taken down and replaced while most subscribers can continue calling each other.

C. Architectural Considerations

The exchange is implemented by decomposing it into a number of similar exchanges, each handling a partition of the set of subscribers. This requires a system architecture, which is a set of structures for reasoning about a system: it does not simply explain how to wire up the individual constituent systems, but rather it explains the consequences of doing so. We want to re-use each exchange without modification, and deduce that each exchange continues to provide its existing service without interfering with additional behaviour added to extend the collective capabilities. To do this, we use a particular architectural pattern called enslavement.

To explain this pattern, consider two systems P and Q, where every external event of P is shared with Q, so that P is entirely controlled by Q, but that Q may do other things. P is then Q's slave: P can communicate with no one other than its master Q. Suppose further that P uses two channels in and out. In our example, Q has many similar slaves—the telephone exchanges—so Q communicates with P using channels labelled by some name, say s: the channels would then be s.in and s.out. So there's an asymmetry between the two of them: Q has to be aware of P's identity (the use of the label s), but P does not need to know who Q is.

We use this architectural pattern in our SoS, where we have a collection of identical copies of the single exchange. They differ only in the set of subscribers they are connected to: they serve different customers, and they do not communicate with each other. They form an unconnected group of systems whose behaviours are merely interleaved. We then put a transport layer above this group. Here, all subscribers talk to a single process, which then relays signals from subscribers to the right exchange slave. Similarly, when the slave responds, its signals are relayed back to the subscriber.

In the first model, all subscriber and call information is centralised, but this new architecture distributes this to local exchanges, and correctness can be proved as a refinement. This is necessary, but does not achieve a geographical separation, since the transport layer is a single service. So we decompose the transport layer into a number of nodes, one per exchange. A subscriber now connects to its nearest local node, which either relays messages to the local exchange or sends something to another node to cause it to interact with its local exchange. This is similar to trunk signalling in telephony, and must provide the same service as the centralised description. A suitable architecture for this transport system is to arrange the nodes in a ring, or to take advantage of geographical considerations to arrange them hierarchically, and we adopt the former.

In the next two sections, we use CML to demonstrate two example scenarios: the first shows the exchange of messages involved in a telephone call using a single exchange. The second shows the start of a call involving two separate exchanges.

D. Single Exchange

Suppose that `Jim` wants to phone `Ana` and that both of these subscribers are linked to a single telephone exchange. We assume in this example that `Jim` is not engaged in any other telephone call. He starts the call by sending a message to the exchange: `call(Jim,Ana)`. This and the subsequent sequence of messages are displayed in Figure 1, which contains a SysML sequence digram for the entire call. Once the exchange has received this call, it acknowledges it by replying with a `callok` message. Internally, the exchange now creates a record to keep track of the call, which is in the `connecting` state, and the identity of the subscriber to whom `Jim` is connecting, `Ana`. The exchange also knows `Ana`'s state; we assume for this example that `Ana` is not busy. The exchange sends the message `startringing` to `Ana`; if `Ana` had been busy, then the exchange would have instructed `Jim`'s equipment to start the engaged tone. After a while, `Ana` answers the call by sending an `answer` message to the exchange, which then acknowledges this and instructs `Jim`'s equipment to start receiving speech packets (which are not described in this model, as they are not part of the telephone signalling protocol). A little while later, `Ana` hangs up. As she is the recipient of the call, she does not own it, and so hanging up suspends the call rather than clearing it, and so `Ana` sends a `suspend`

message to the exchange, which acknowledges it. A little while later, Ana picks up the phone again and re-establishes the call by sending an unsuspend message to the exchange, which again acknowledges it. Finally, Jim hangs up, clearing the call, since he owns it as the call initiator. This clear message causes the exchange to delete the call record and start a new one for Ana to register the fact that she is still off-hook. When she eventually hangs up, then that final fragment of the call can be deleted.

This informal description of the telephone call, and the accompanying SysML sequence diagram, are based on a mathematical model in CML. As well as explaining what is going on, the model can also be used to predict events and situations: 1) Is it ever possible for the telephone exchange to deadlock? 2) Given the real-time properties of the various components, what are the response and connection times? 3) Are all connected telephone calls one-to-one (no sharing)? 4) What happens when Jim calls himself? 5) What happens when Ana calls him during one of his self-obsessed calls?

We give just a flavour of the CML specification of the exchange. It has three instance variables:

```
public status: map SUBS to STATUS;
public number: map SUBS to SUBS;
public subs: set of SUBS;
```

The status and number variables record information for each call in progress; subs records the set of subscribers linked to this exchange; status is a mapping from subscribers to the status of their calls, such as connecting, speech, or ringing, while number is a mapping from subscribers to subscribers. These two mappings record all necessary information, so an invariant requires that they record the same set of subscribers. In our example, number maps Jim to Ana from the point at which Jim initiates the call until he clears it down. The call passes through the sequence: connecting, ringing, speech, suspended, and speech (again), before being deleted.

Each message received by the exchange triggers a state operation. For example, a new call is handled in the following way, specified in VDM:

```
Call (s,t: SUBS)
frame wr status, number
      rd subs
pre
  s in set subs and
  t in set subs and
  s in set free(status,number,subs)
post
  status = status~ ++ {s |-> <connecting>} and
  number = number~ ++ {s |-> t};
```

The Call operation takes two subscribers s and t as parameters, and has write access to the status and number instance variables and read access to the subs variable. The precondition requires that s and t belong to the set subs, which means that they are subscribers linked to this exchange. Additionally, the precondition requires that s is also free. The function free returns the set of subscribers who are neither initiators nor recipients of calls in progress. The postcondition describes the effect on the instance variables, which are updated to record the new call, which is between s and t and is in the connecting state.

The Call operation describes the changes in the internal state of the exchange that take place following the reception of a call message. The reactive aspect of the operation is specified in CSP. Here, we see the first part of the behaviour in the definition of the Exch process:

```
Exch =
  call?s:(s in set subs)?t:(t in set subs) ->
    ( if s in set free(status,number,subs)
      then callok -> Call(s,t)
      else callerror -> SKIP ); Exch
...
```

A request can be received, via the call channel, to establish a connection between two subscribers, s and t, both of which are linked to this exchange. Following the reception of this message, the exchange tests to see if s, the initiator, is actually free. If it is, then a positive acknowledgement is sent (callok), and the state operation Call(s,t) (above) is invoked. Otherwise a negative acknowledgement is sent (callerror). After this, the process repeats its main loop.

E. Multiple exchanges

Figure 2 describes the interactions between participants in an SoS of just two telephone exchanges, using a SysML sequence chart. We assume in this very simple instantiation there are only two clusters of subscribers, one in York and one in Aarhus. A subscriber in York, Jim, wants to put a call through to a subscriber in Aarhus, Peter. Jim initiates the call with the message call(Jim,Peter), which the SoS receives at the York node. York detects that this call is not to another subscriber in York, but to someone in Aarhus. So it makes a virtual call to Aarhus and sends a vcall message along the

mid channel to the York link. This is the process responsible for all messages going out on the ring from York; it can buffer messages up to some limit. The link then forwards this message to Aarhus, the next node in the ring. Aarhus detects that this message is indeed destined for Aarhus, and so takes it off the ring. Aarhus now completes the virtual call by first adding Jim as an Aarhus subscriber and then making a local call from Jim to Peter. The Aarhus exchange responds with an acknowledgement that the call is in progress. It will not have been connected yet, since the exchange needs to find out if Peter is free or busy, but that will follow. In the meantime, the Aarhus node needs to relay this acknowledgement back to Jim in York, so it sends the acknowledgement along its mid channel to the Aarhus link. The link then forwards this along the ring, with the York node recognising and intercepting the message, which can finally be delivered to Jim.

Again, this model can be analysed to prove various properties. Unlike the centralised single exchange, the SoS is a distributed system and the problems of deadlock and livelock are more acute. For example, the links must have a bounded amount of buffering, so what happens when buffers fill up: does the system deadlock? Messages pass around the ring: is it possible for a message to be perpetually travelling? Or we could ask if the SoS of exchanges provides the same functionality as a single exchange would. That is, is the SoS a refinement of the single exchange? If it is, then it has all the properties that the single exchange has: functionality, liveness, real-time performance, and so on. This can be checked using a refinement model checker. This kind of tool would explore all the behaviours of the SoS and make sure that they were all behaviours of the single exchange. For this to make sense, CML provides a way of hiding certain channel communications from view. For example, the communications between constituent nodes are not part of any behaviour in the single exchange.

A model checker has two very strong advantages: it is a push-button, fully automatic tool; and when the model checking fails (as is most often the case in debugging a specification), it produces an explicit counterexample that may be useful is working out why the refinement does not hold. But there is a drawback: there is a limit to the number of behaviours that can be checked in this way, and so the size of the CML models becomes an issue. This can be overcome by using a symbolic model checker that replaces specific values by symbolic names and proceeds to check whole classes of behaviours at once. Such tools can even check infinite behaviours. An alternative to model checking is theorem proving, which can in principle prove properties regardless of the number of behaviours.

4. Conclusions

Our main contribution in this work is in the design of a language specifically for modelling and reasoning about SoSs. It is suitable for describing both constituent systems and the architecture needed for composing them into an SoS. The formal basis of this language is suitable for building powerful analysis tools for verification and validation.

The complete development of CML—its syntax, semantics, refinement technique, connection to SysML, and tools platform—is the subject of ongoing work in COMPASS. At the moment, given the wealth of experience available in the use of VDM, CSP, Circus, and their semantics in the UTP, we are confident of the soundness of our approach. In particular, we can rely on the mature tools of VDM and CSP for restricted reasoning while the CML tools are under development [1].

References

[1] J. W. Coleman, A. K. Malmos, P. G. Larsen, J. Peleska, R. Hains, Z. Andrews, R. Payne, S. Foster, A. Miyazawa, C. Bertolinik, and A. Didier, "COMPASS Tool Vision for a System of Systems Collaborative Development Environment," in The 7th International Conference on System of System Engineering, IEEE SoSE 2012, July 2012.

[2] J. Holt and S. Perry, SysML for Systems Engineering. IET, 2008.

[3] J. Woodcock and A. Cavalcanti, "The semantics of Circus," in Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B, ZB '02, (London, UK, UK), pp. 184–203, Springer-Verlag, 2002.

[4] M. Spivey, The Z Notation – A Reference Manual (Second Edition). Prentice-Hall International, 1992.

[5] T. Hoare, Communication Sequential Processes. Englewood Cliffs, New Jersey 07632: Prentice-Hall International, 1985.

[6] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef, "Vienna Development Method," Wiley Encyclopedia of Computer Science and Engineering, 2008. edited by Benjamin Wah, John Wiley & Sons, Inc.

[7] T. Hoare and H. Jifeng, Unifying Theories of Programming. Prentice Hall, April 1998.

[8] C. B. Nielsen and P. G. Larsen, "Extending VDM-RT to Enable the Formal Modelling of System of Systems," in The 7th International Conference on System of System Engineering, IEEE SoSE 2012, July 2012.

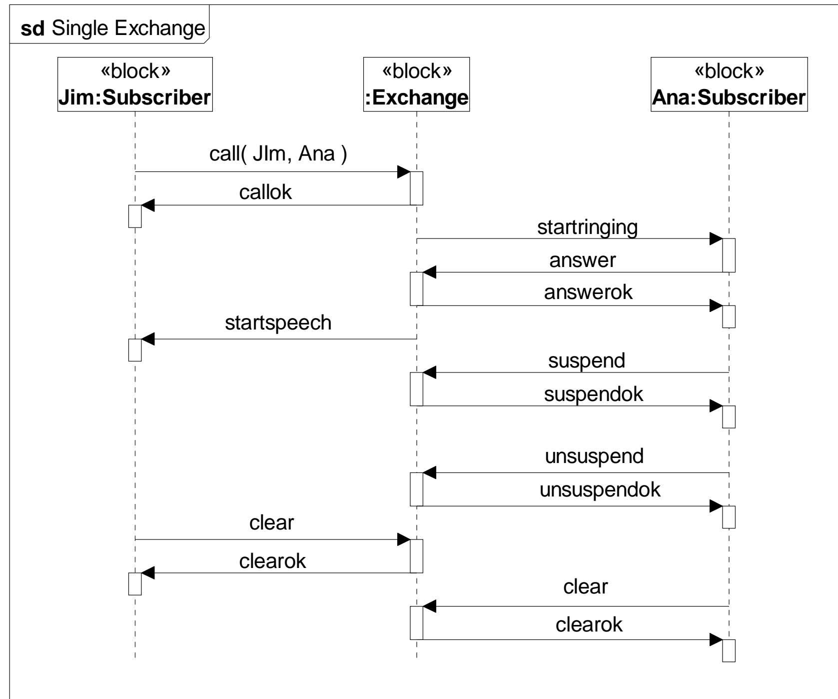

Figure 1. Telephone call in a single exchange.

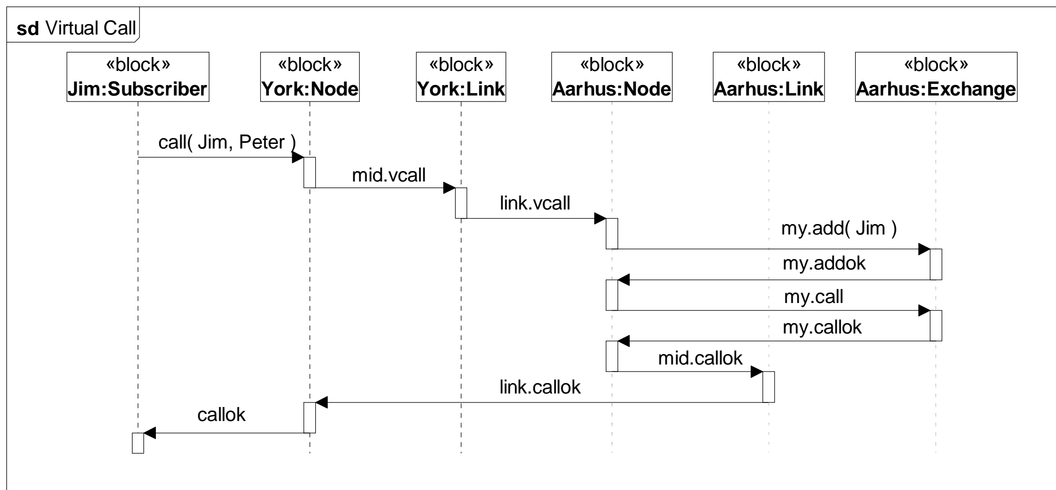

Figure 2. Virtual call.