COMPUTING SCIENCE

Semi-Formal and Formal Interface Specification for System of
Systems Architecture

Jeremy Bryans, Richard Payne, Jon Holt and Simon Perry

TECHNICAL REPORT SERIES

No. CS-TR-1371

February 2013

Semi-Formal and Formal Interface Specification for System of Systems Architecture

J. Bryans, R. Payne, J. Holt, S. Perry

Abstract

Provenance management still lacks robust models for sharing provenance data between multiple parties while keeping parts of it private to the owner. This limits the potential for provenance dissemination, which is a critical step in enabling data sharing amongst partners with limited a priori mutual trust. In turn, this has a negative impact on data-intensive science and its associated research publication repositories, on audit tasks, as well as on increasingly common collaborative dynamic coalitions scenarios. We propose a method for preserving privacy by creating abstractions over provenance graphs, we apply it to provenance sharing, and illustrate it on a health care case study.

Bibliographical details

BRYANS, J., PAYNE, R., HOLT, J., PERRY, S.

Semi-Formal and Formal Interface Specification for System of Systems Architecture
[By] J. Bryans, R. Payne, J. Holt, S. Perry

Newcastle upon Tyne: Newcastle University: Computing Science, 2013.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1371)

Added entries

NEWCASTLE UNIVERSITY

Computing Science. Technical Report Series. CS-TR-1371

Abstract

The independence of the constituent systems of a system of systems presents a key challenge to the discipline of system of systems (SoS) engineering. The fact that constituent systems can and do function independently of the SoS means that engineers of a constituent system cannot rely on the behaviour of other constituent systems. This paper advocates a model-based approach to SoS engineering that requires the interfaces to constituent systems to be specified. We propose an use of an interface design pattern for interface specification that uses the industry standard notation, SysML. We also indicate a translation of these specifications to a formal notation, CML, in order to extend the range of analytic techniques available to the SoS engineer.

About the authors

Jeremy is a Senior Research Associate in the School of Computing Science, and a member of Centre for Software Reliability. His research interests are in the modelling and analysis of collaborating systems, and in the development of trustworthy policies for their interaction. He is currently a Co-Investigator on the EPSRC project Trusting Dynamic Coalitions, on which he works on designing and composing provenance policies for coalition members. Part of his time is spent on the EU project COMPASS, on which is developing semantic foundations for a modelling language for Systems of systems.

Richard Payne obtained his PhD in 2012 at Newcastle University under the supervision of Prof. John Fitzgerald, titled Verifiable Resilience in Architectural Reconfiguration. As part of his PhD, Richard provided a basis for the formal verification of policies defined using a reconfiguration policy language (RPL) for the governance of resilient component-based systems. Richard worked as an RA on the Ministry of Defence funded SSEI project and was involved in the 'Interface Contracts for Architectural Specification and Assessment' sub task, investigating the use of contract-based interface specification in system of systems architectural models. Richard is now working on the COMPASS project, on the use of model-based techniques for developing and maintaining systems of systems, involved with work in architectural modelling, fault modelling and tool development.

Jon Holt is currently the Global Head of Systems Engineering for Atego, a leading independent provider of tools and capability for systems engineering, and was the founder-director of Brass Bullet Ltd, a systems engineering consultancy and training company for over 12 years, until it was acquired in 2009. Jon is also a Professor of Systems Engineering at the UK Defence Academy, where he is involved with teaching and research. Jon is active in the IET via their Professional Networks and the BCS as a member of the Learned Society. He is a Fellow of both the IET and the BCS and is a Chartered Engineer and Chartered IT Professional. Jon is an international award-winning author and public speaker in the field of applied systems engineering and research. He has authored nine books on systems engineering. His books cover the application of UML and SysML to systems engineering, process modelling, enterprise architectures and competency assessment. His main area of interest is the application of systems modelling to all aspects of system engineering and his areas of expertise include: UML and SysML for system engineering, process modelling, standards compliance, requirements engineering, life cycle modelling, enterprise architectures, architectural frameworks and competency assessment.

Simon Perry holds Bachelor degrees from both the University of Leeds and the Open University. Since gaining his degree in Mathematics in 1986 he has spent over 25 years working in all aspects of software and systems engineering. Simon often speaks at systems engineering conferences and is the co-author of five books on systems engineering and related topics: 'SysML for Systems Engineering', 'Modelling Enterprise Architectures', 'Model-Based Requirements Engineering' and 'Model-based Systems Engineering using SysML' published by the IET and 'A Pragmatic Guide to Competency' published by the BCS. Simon is a Principal Consultant for Atego Systems, providing consultancy, training and conducting research in the application of systems engineering. He works in industry, government and academia and has applied his work across many disciplines in a wide range of industries including defence, the nuclear industry, timber engineering, finance and train manufacture. He is a Member of the IET, INCOSE and is a part-time lecturer at the University of Warwick.

Suggested keywords

SYSTEMS OF SYSTEMS
ARCHITECTURE
SYSML
CML
INTERFACE SPECIFICATION
CONTRACTS

Semi-Formal and Formal Interface Specification for System of Systems Architecture

Jeremy Bryans¹, Richard Payne¹, Jon Holt², and Simon Perry²

¹Newcastle University, UK. {jeremy.bryans, richard.payne}@ncl.ac.uk

²Atego, UK, {jon.holt, simon.perry}@atogo.co.uk

April 11, 2013

Abstract

The independence of the constituent systems of a system of systems presents a key challenge to the discipline of system of systems (SoS) engineering. The fact that constituent systems can and do function independently of the SoS means that engineers of a constituent system cannot rely on the behaviour of other constituent systems. This paper advocates a model-based approach to SoS engineering that requires the interfaces to constituent systems to be specified. We propose an use of an *interface design pattern* for interface specification that uses the industry standard notation, SysML. We also indicate a translation of these specifications to a formal notation, CML, in order to extend the range of analytic techniques available to the SoS engineer.

1 Introduction

Systems of systems (SoS) engineering faces two significant challenges. On the one hand, the managerial and operational independence of the constituent systems within an SoS places limits on the range of design options available to the engineer, who must seek to develop an SoS bounded by the behaviour of the extant constituent systems. On the other hand increasingly complex SoSs are being developed, and society is placing more and more reliance on them. Together, these necessitate efficient methods, techniques and tools to assure key properties (such as safety properties) of SoSs. The INCOSE Systems Engineering Handbook [26] also identifies challenges in the development of SoS including: constituent system independence, ambiguous requirements, complexity and difficulty in defining SoS boundaries.

Although independence offers unique challenges to SoS engineering, we take the view of Calinescu and Kwiatkowska [5], in proposing the adaption of current methods and tools as starting points rather than supplanting current best practice. As such, we believe one method for tackling the problem of independence is the description of SoS architectures. In particular, a promising method for SoS architectural definition lies in *interface specification*, whereby the independence of constituents is explicitly acknowledged and respected. These then

provide the basis for a contractual relationship between the constituent systems and the SoS. Interface specification of an SoS architecture allows the internal definition of constituent systems to change, as long as it continues to respect the contractual interface specification.

In this paper we advance previous work [27], where we proposed the approach of defining interfaces of a SoS architecture using SysML [25] in terms of publicly-visible operations, and modelling a corresponding formal model using the formal specification language VDM [9]. There we concluded that, while the approach was a useful way to deal with some issues resulting from independence, there were elements required of a formal notion which VDM alone could not provide.

Here we take the same broad approach. We again use SysML, an established semi-formal¹ notation with a strong history of use in industry, to model the SoS architecture. In previous work, the interface modelling covered the identification of interfaces and definition in terms of a collection of operation signatures.

The contribution of the paper is to demonstrate a model-based approach to interface specification for SoS architectures, using the industry standard notation, SysML and a formal notation CML [30]. We demonstrate the use of the novel *interface design pattern* [31] to specify both the structure and behaviour of interfaces in SoS architectures in the industry standard notation, SysML. This interface definition is translated to a formal notation, CML, which provides both data modelling and event ordering to develop a corresponding formal model of the SoS architecture and interfaces.

The remainder of the paper is structured as follows. In Section 2, we consider related work in the fields of SoS, architecture and interface specifications, and formal modelling. Section 3 outlines our approach by means of a simple illustrative example, defining interfaces using SysML and CML. We evaluate the approach in Section 4 and offer some conclusions in Section 5.

2 Related Work

We introduce basic concepts and terms used in our work relating to SoS, discuss approaches to specifying SoS architectures and contracts, and finally give more detail on the notations for formal model specification.

2.1 Systems of systems

SoS are network-enabled integrations of heterogeneous systems delivering capabilities and services which result in emergent behaviour which cannot be achieved by the constituent systems alone [18, 3]. Maier distinguishes SoS from large monolithic systems by several characteristics, and a literature survey [21] identified eight key ‘dimensions’ of SoS, based on a wide examination of the literature characterising SoSs. These eight properties of an SoS are: autonomy, independence, distribution, evolution, emergent behaviour, dynamic behaviour, interdependence and interoperability.

Attempts have been made to classify SoS by the level of managerial control placed upon them [6]. The types – *Virtual*, *Collaborative*, *Acknowledged* and

¹We consider a semi-formal notation to have a defined syntax, but without a mathematically sound semantics. A formal notation has both a defined syntax and mathematically sound semantics.

Directed – are distinguished largely by their central authority and collaboration in evolution.

The goal of our work, supported by the COMPASS project², is to develop modelling languages that are expressive enough to model the architecture and behaviour of candidate SoS structures, and sufficiently rigorously defined to permit trustworthy machine-assisted analysis of global properties.

2.2 SoS Architecture and Interface Specification

The majority of work on architecture description languages (ADLs) is at the level of software architecture [19]. Many existing notations (including formal ADLs such as Darwin [17]), therefore, do not contain architectural abstractions suitable for modelling SoSs, such as the notions of SoS or system. The advancement of model-based approaches to embedded systems has strengthened the need for notations modelling both hardware and software elements. UML [24] may be used in systems engineering [13], and recent notations including SysML [13, 25] and AADL [7] explicitly address architectures at the system engineering level and may be considered for SoS descriptions [28].

Within architectural description notations, the state of the art in interface specification is limited [28]. The most widely used notations (UML [24] and SysML [25]) allow basic signatures to be defined and pre- and postconditions to be specified textually, but these are rarely used in practise. In AADL [7] models are defined in terms of component types and implementations which include subprograms (similar to operations) with basic signatures, though pre- and postconditions are not available. Formal architectural notations such as Darwin [17] and Wright [1] allow the definition of ports for software components, and (in the case of Wright) to have their message exchange protocols defined. However, these notations do not include the ability to specify other details such as operation pre- and postconditions.

2.3 Contracts

We propose the *contractual* description of architectural interfaces, in terms the assumptions they rely on and the obligations placed on their behaviour [28]. This has much in common with the idea of Design by Contract (DbC), a software engineering technique for object-oriented software introduced by Meyer [20] in which contracts make explicit the functional relationships between systems in terms of preconditions and postconditions on operations and invariants on states. The use of DbC in software allows designers to define the expected interfaces of operations, which may then be provided to the engineers. This provides greater confidence to designers that operations will adhere to the expected properties on interfaces. The use of component-based software techniques is key in system integration. System integration patterns rely on the composition of software components with contractually defined interfaces [12].

The use of contracts is an active research field in service-oriented computing (SOC), including web services where contracts are defined largely using XML-based languages including WSDL [32]. Service contracts typically include func-

²Our work focuses on dependability and formal model-based design methods for SoS in the COMPASS project (Comprehensive Modelling for Advanced Systems-of-Systems, EC FP7 Project 287829, www.compass-research.eu).

tional descriptions of service functions and service-level agreements (including non-functional requirements). Beugnard et al. [2] propose a four-level contract for service-oriented architectures, including the scheduling of component interaction and message passing as well as non-functional aspects of operations. Contracts are subscribed to prior to service invocation, after a period of negotiation. Orchestration (WS-BPEL [22]) and choreography (WS-CDL [33]) notations for services have been standardised, interacting with external entities through service contracts defined using WSDL. Formal treatments of SOC notations have been attempted, such as a pi-calculus based semantics for WS-BPEL [16] and the use of formal methods in web services is a topic attracting much interest (see, for example, the workshop series on Web Services and Formal Methods (WS-FM)³).

In previous work, we consider the specification of non-functional properties in contractually defined architectural interfaces [28], which we also consider in [31].

2.4 Formal modelling

Formal methods can be challenging, but advances in tool support and machine-assisted analysis have greatly increased their potential for use in industry software engineering [36, 8]. These techniques, however, have been applied only experimentally in SoS engineering (e.g. [4]). There are many formal languages suitable for expressing and analysing particular features of a system. It is apparent, however, that for a SoS a language needs to cover aspects such as functionality, concurrency, communication, inheritance and time.

The baseline for our work on formal modelling is a combination of two formal notations that together provide the features that we require: Circus and VDM. Circus [35] combines CSP, Z, specification statements and guarded commands to provide a language for specification, programming, and verification by refinement. The Circus model is a UTP theory that already combines the theory for contracts, and the theory for concurrency. VDM [9] is a modelling language for state-based systems, integrated with object-oriented and real-time constructs, and supporting descriptions of data and functionality in the form of executable code or as contracts.

3 Modelling approach

We present the proposed modelling approach using a simple example based on a travel agent SoS. This is an SoS in Maier's terms [18] and may be considered an *acknowledged* SoS [6].

The travel agent example is based on the simple and well-known travel agent problem as described in the DSoS project [23], an EU project on dependable SoS running from 2000 to 2003. The SoS consists of a travel agent front end linked to a number of travel booking systems (booking systems). The SoS aims to bring together a number of flight and hotel booking systems to provide a SoS that will allow users to request and book entire trips. Each constituent system may include human elements (for example, a hotel booking system could be a person with minimal computing abilities).

³<http://www.informatik.uni-rostock.de/ws-fm2012/>

A trip service is provided, which allows the user of the travel agent SoS to enter the destination and the period of the trip. The task of the front end is to retrieve the information from the various constituent booking systems, assemble it, and present it to the user in the form of full trips. In order to carry out such a task the travel agent SoS has to manage a number of transactions between the front end, booking systems and the SoS user.

When describing the SoS architecture in the subsequent section, we pay particular attention to the interfaces between the different constituent systems and describe the different operations offered as well as the operation choreography.

We offer an architectural description in SysML in Section 3.1, and in particular give details on one of the interfaces between the systems. Section 3.2 supplements this interface description with a formal definition in CML.

3.1 SysML

SysML [25] is a profile for UML 2.0, originally developed for system engineering, and which supports the modelling of SoS architecture. It enjoys wide industrial support and a sound tool base. SysML provides several diagram types and has a “precise natural language” semantics, to support the description of the SoS structure, behaviour and requirements.

The modelling approach we employ is to define the SoS boundary in terms of its constituent systems. The SoS contains several constituent systems: a front end which provides an interface to the SoS, a central flight booking system, several booking systems and booking system wrappers. We identify these initially within a SysML block definition diagram given in Figure 1.

Figure 1: Block definition diagram depicting travel agent SoS structure.

Following the identification of the constituent systems, we focus on the definition of the interfaces exposed between the constituents and between the SoS and its environment. To describe each interface we provide a set of views, given as SysML diagrams. These views are described for an interface identified for the example in Section 3.1.1.

3.1.1 The Interface Design Pattern

The *interface design pattern* is defined to be a set of views given by SysML diagrams [31]. The views in the interface pattern define the interface connections, contents and behaviour and include the *interface connectivity view*, *interface definition view*, *interface behaviour view* and *protocol definition view*.

The composition of these constituent systems is described in the interface connectivity view given in Figure 2, in which the interfaces are explicitly identified. This identifies the provided and required interfaces of the constituent systems of the SoS. For example, the interface *IF_CentralFlight* is provided by the central flight booking system and required by the front end system.

Figure 2: Interface Connectivity View showing relationships of constituents.

The interface definition view, based on the block definition diagram, defines the operations on an interface, and the datatypes used by the interface. The interface behaviour view, based on the SysML sequence diagram, provides scenarios to illustrate possible means of using the interface between the participants of interactions. Finally, the protocol definition view, based on the SysML state machine diagram, defines the permitted ordering of operation calls on an interface. The final view is similar in nature to the UML protocol state machine diagram [24], which was omitted from the definition of SysML.

We illustrate the interface design pattern using the *IF_CentralFlight* interface, as shown in Figure 2. The remainder of the SysML model containing definitions of the other interfaces, may be found in Appendix A.

Figure 3 depicts the interface definition view for *IF_CentralFlight*. In this case, the interface has six operations, with their parameters and return types defined. For each type, a *data type* block is given with their contents also defined. The SysML specification allows pre- and postconditions to be specified for operations on interfaces, however this is optional and no analyses are available to ensure their correctness.

Given the operations defined for an interface, we consider scenarios of possi-

Figure 3: IF_CentralFlight Interface Definition View.

ble interactions between the provider and user of the interface. Several interface behaviour views are defined for the case study in the form of SysML sequence diagrams, which depict a sequence of calls to the operations of an interface. Figure 4 depicts one such view, in which a transaction ends due to a timeout. More specifically, the view shows that the front end system requests flight availability, then retrieves prices for a flight and places a reservation for seats on the flight. After a timeout, the seats are released. In this example, each operation call receives some result.

The final view is the protocol definition view, which depicts a state machine of the permissible behaviour of the interface. This is given using a SysML state machine diagram. Figure 5 gives the protocol definition view of the IF_CentralFlight interface. This depicts an “execution” of the interface, and has an initial state, two intermediate states and a final one. Once initialised, the interface is in the intermediate *Ready* state. From this state, calling the *flightPrice* or *flightAvail* operation returns the interface to the ready state. Alternatively, the *reserveFlight* operation may be called resulting which results in the state machine moving to the intermediate *Reserved* state. From this state, the reserved seats may be released or booked. Finally, a flight may be booked directly from the *Ready* state without entering the reserving seats state, using the *bookFlight* operation.

SysML omits the protocol state machine of UML 2.0 which may be of use when defining the protocol definition view – allowing the use of pre- and postconditions on the transitions between states. The difference between the protocol state machine and SysML state machine diagrams is subtle, and relates to the semantics of the diagram transitions. This is discussed in more detail in [29]. It is our opinion that this construct would add additional rigour to an interface

Figure 4: IF_CentralFlight Interface Behaviour View.

Figure 5: IF_CentralFlight Protocol Definition View.

specification and increase the range of analyses available.

Given a full specification of all the interfaces of the SoS, we may consider some examples of complete SoS behaviour. In Figure 6, the SysML sequence diagram shows the sequence of interface operation calls between the various constituent systems involved in booking hotel and flights for a trip (we omit

the booking systems for readability). It should be noted that the sequence of operation calls is consistent with the IF_CentralFlight interface protocol definition and uses a subset of the operation calls of the interface behaviour view in Figure 4.

Figure 6: SysML Sequence Behaviour Diagram showing complete SoS behaviour for booking and reserving flights and hotel.

At the point at which a reservation is made, there are two possibilities for progress in the IF_CentralFlight interface protocol definition. The first, shown in the interface behaviour view in Figure 4, follows a timeout – modelling the case where the user does not proceed with the booking. Figure 7 shows the parallel release of flight and hotel bookings.

The other case occurs when a booking is completed, as described in Figure 8. This describes a different case to the one described in the interface behaviour view in Figure 4, although it is consistent with the IF_CentralFlight interface protocol definition in Figure 5.

The remainder of the SysML model containing definitions of the other interfaces using the interface design pattern which may be found in Appendix A. As the purpose of this is to exercise the interface design pattern, and not to accurately model individual booking systems, we do not model in detail the

Figure 7: SysML Sequence Behaviour Diagram showing complete SoS behaviour for timeout of reserved flights and hotel.

Figure 8: SysML Sequence Behaviour Diagram showing complete SoS behaviour for paying for reserved flights and hotel.

individual hotel booking system interfaces (IF_HBS1 and IF_HBS2).

3.2 CML

CML is the *COMPASS Modelling Language*, and is the first language to be designed specifically for the modelling and analysis of Systems of Systems [34]. It is based on the languages VDM [9], CSP [10] and Circus [35]. 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. Using CML, many different kinds of analyses can be conducted, and some will be presentable at the SysML level. A semantic model for CML using UTP [11] is in development.

In this approach, we build each interface in CML, using the interface de-

defined in the interface definition views, and the behaviour defined in the protocol definition views. The SoS specification is given below by the interleaving combination of these interfaces (denoted by the `|||`) operator, which for simplicity we assume do not share events.

```
process SoS = IF_TripBooking ||| IF_CommonHotel |||
              IF_HBS1 ||| IF_HBS2 |||
              IF_CentralFlight ||| IF_CommonFlight
```

To give a flavour of the CML representation of an interface, consider the specification of the central flight interface: *IF_CentralFlight*. The CML process representing the interface is outlined below. The CML process contains local **state**, a collection of **operations** and **actions**, which describe the permitted protocol of the interface.

```
process IF_CentralFlight =
begin
  state
  ...
  operations
  ...
  actions
  ...
  @
end
```

The approach we take is to specify the external behaviour of the interface. From the SysML interface definition (Figure 3) we see that the **flightAvail**, **flightPrice**, **reserveFlight**, **bookReserved**, **releaseFlight** and **bookFlight** operations are present. These are added to the above process definition. We also add pre- and post conditions to the operations. For example, consider the **flightAvail** operation. The precondition uses a **laterDate** function to ensure that the end date on the flight request is later than the start date. The post-condition requires that all returned flights are on the required date, fly to the correct destination and have the required seats available. Also, notice that some operations (for example **bookReserved**) use private state variables to represent the collection of reserved and booked flights.

```
operations
public flightAvail(req:FlightReq) fl:set of Flight
pre laterDate(req.startDate, req.endDate)
post forall f in set fl @
  ((f.flightDate = req.startDate) and
   (f.dest = req.toLoc) and
   (f.seatsAvail >= req.numPassengers))

public flightPrice(fl:Flight, n:nat) p:Price
pre fl.seatsAvail >= n
post p > 0

public reserveFlight(f:Flight, n:nat) r:ReserveId
pre f.seatsAvail >= n
post r in set reservedFlights
```

```

public bookReserved(r:ReserveId) fbId:FBookingId
pre r in set reservedFlights
post fbId in set bookedFlights

public releaseFlight(r:ReserveId)
pre r in set reservedFlights
post r not in set reservedFlights

public bookFlight(f:Flight, n:nat) fbId:FBookingId
pre f.seatsAvail >= n
post fbId in set bookedFlights

```

From the Protocol Definition View (Figure 5) we see that there are two main states: `CF_Ready` and `CF_Reserved`. The CML segment below defines CML actions corresponding to the two states, with the main initial action (denoted by the `@` symbol) being the `CF_Ready` state.

```

actions
  CF_Ready  =

  CF_Reserved =

  @ CF_Ready

```

The fragment of CML below describes the `CF_Ready` state. The *Front End* constituent system does not call the operation directly. Instead the request is communicated through a collection of event channels: `reqFlightAvail`, `reqFlightPrice`, `reqReserveFlight` and `reqBookFlight`, with results communicated through corresponding `return*` channels. As there is no restriction on ordering on the environment of the interface, the external choice operator `[]`, separates the choices. Considering the first case, `reqFlightAvail`, a temporary variable `s` reads the output from the `flightAvail` operation (which will be a set of flights) and communicates this to the front end, in this case via the channel `returnFlightAvail`, before returning to the state `CF_Ready`. Alternative available behaviours in the `CF_Ready` state include access to `flightPrice`, `reserveFlight` and `bookFlight` operations, which are handled similarly.

```

actions
  CF_Ready  =
    reqFlightAvail?flreq ->
      (dcl s : set of Flight @ s := flightAvail(flreq);
       returnFlightAvail!s -> CF_Ready)
    []
    reqFlightPrice?fl?num ->
      (dcl p : Price @ p := flightPrice(fl,num);
       returnFlightPrice!p -> CF_Ready)
    []
    reqReserveFlight?fl?num ->
      (dcl r : ReserveId @ r := reserveFlight(fl,num);
       returnReserveFlight!r -> CF_Reserved)
    []
    reqBookFlight?fl?num ->
      (dcl fbId : FBookingId @ fbId := bookFlight(fl,num);
       returnBookFlight!fbId -> CF_Ready)

```


Once a reservation has been made, and the `CF_Reserved` state reached, the *Front End* system can request that the reservation is booked, in which case the interface reaches the final state. A timeout is triggered if this takes too long, in which case the reservation is released and the interface returns to the `CF_Ready` state. This is shown in the `CF_Reserved` action of the CML model, below. The $P \ [(\mathfrak{t}) > Q$ operator behaves as P until a timeout $\mathfrak{t}$, then as Q . In this case, the timeout is denoted by a private local variable of the *IF_CentralFlight* process – `timeout`.

```
actions
CF_Reserved =
  (reqBookReserved?rId ->
    (dcl fbId : FBookingId @ fbId := bookReserved(rId);
     returnBookReserved!fbId -> CF_Ready))
  [(timeout)>
    unreservedFlight?r -> releaseFlight(r); CF_Ready
  ]
@ CF_Ready
```

We apply this approach to the other interfaces in the architecture (Figure 2), presented in full in Appendix B.

4 Evaluation

In this section we make comparisons to the interface and contract specification areas of related work identified in Section 2.

Our approach to semi-formal interface definition builds upon existing the SysML notation through the use of the *interface design pattern*. This is strengthened by the translation to the formal CML interface description to make explicit the operation pre- and postconditions, and the event ordering. This advances the state of the art in interface specification – through the use of the approach to semi-formal interface specification and translating to the analysable formal description.

With respect to contract specification, we consider the three approaches identified in Section 2.3; Design by Contract, service contracts and integration patterns. The DbC technique, proposed for software engineering, proposes similar concepts as those identified in this report. The approach we propose is inspired by this thinking, though is lifted to the SoS-level, and incorporated into the CML language so to enable formal analysis of SoS models. Whilst not considered in this paper, CML includes the use of type and instance variable invariants, as proposed in DbC. These properties describe properties that must hold over all behaviours of the SoS. Where DbC is limited to the operations of software classes, our approach also includes the use of the semi-formal protocol definition view and the formal CML process definition.

Whilst there have been formal treatments of the largely XML-defined service contracts, the approach in this paper provides a rigorous SoS engineering method for contractually described interfaces, which enable formal analysis. Our approach, therefore, allows non-formalists to define SoS models with an existing industry-standard notation, and translate these to the CML formal notation for analysis. We may, however, consider applying the service-oriented architecture

pattern in SoS design. As such, considering the relationship between constituent system interfaces and service interfaces is useful, including the underlying Service Contract concepts. An initial report on our investigations of this may be seen in [31]. Although we propose the SysML-CML link in this paper, we could consider translating the XML-based notations used in Service Contracts (for example WSDL and WS-BPEL) to CML so as to take advantage of the CML analysis techniques.

The experience of enacting this approach which the travel agent case study has given us some insight into some weaknesses of our work. The interface design pattern does not include pre- and postconditions in the operation definitions of the interface definition view, they only appear in the formal CML definition. As such, they should be made explicit in the interface definition view.

The CML modelling demonstrated in this paper is performed manually. Whilst the use of formal methods in industry is increasing [36, 8], this may be seen as a large hurdle for industry up take. As such, an automatic translation is required. This SysML-to-CML translation is currently in development, with automated translation a long-term goal.

5 Conclusions

In this paper we have demonstrated a model-based approach to interface specification, deriving formal models of systems of systems. We modelled a simple SoS using SysML to rigorously define interface functionality and behaviour. Using this SysML model, we defined a corresponding CML model of the SoS. In deriving the CML model, we made use of each of the SysML views. This paper extends our existing work in this field in two ways. The SysML modelling of the interfaces is more complete – we provide views that illustrate the interactions at the interface and that clarify the interface protocols. Secondly, we use the new formal notation, CML, that addresses the requirements for a suitable formal notation set in [27]: the ability to describe data-based specification of functionality and accepted event orderings at the interface.

CML and its tool support are still in development. As such, automated analysis is not yet available directly. Current tool development, however, aims to provide several forms of strong analysis support. This includes testing, simulation, theorem proving and model checking. In the interim, we can take advantage of the analysis available for the languages on which CML is based.

Several areas of further work may be considered. The example in this paper is deliberately simple so not to distract from the approach it illustrates. In future work, we propose the use of this model-based approach in a more complex study. The CML modelling demonstrated in this paper is performed manually using an initial version of the language. Further work in this area would be to use SysML-to-CML translation rules currently in development, with automated translation a long-term goal. Finally, in model-based systems engineering, prior to system design, requirements engineering is essential [15]. An area of future work is to consider interfaces at the requirements stage of SoS engineering [14], and methods of using requirements in context to drive SoS interface specification.

Acknowledgements

The authors' work is supported by the EU FP7 Project COMPASS (<http://www.compass-research.eu>), the UK EPSRC Project Trusting Dynamic Coalitions, and the EPSRC Platform Grant on Trustworthy Ambient Systems (TrAmS-2). The authors are grateful to John Fitzgerald, Alvaro Miyazawa and the anonymous reviewers of the version of this paper submitted to SysCon 2013 for their feedback and comments on earlier versions of the paper.

References

- [1] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.
- [2] Antoine Beugnard, Jean-Marc Jézéquel, Noel Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.
- [3] John Boardman and Brian Sauser. System of Systems – the meaning of “of”. In *Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering*, Los Angeles, CA, April 2006. IEEE.
- [4] J.B. Caffall, D.S.; Michael. Formal methods in a system-of-systems development. In *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, pages 1856–1863. IEEE, October 2005.
- [5] Radu Calinescu and Marta Kwiatkowska. Software Engineering Techniques for the Development of Systems of Systems. In *Foundations of Computer Software. Future Trends and Techniques for Development*, volume 6028 of *Lecture Notes in Computer Science*, pages 59–82. Springer Berlin/Heidelberg, 2010.
- [6] Judith S. Dahmann, George Rebovich Jr., and Jo Ann Lane. Systems engineering for capabilities. *CrossTalk Journal (The Journal of Defense Software Engineering)*, 21(11):4–9, November 2008.
- [7] Peter Feiler, David Gluch, and John Hudak. The architecture analysis and design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, 2006.
- [8] J. S. Fitzgerald and P. G. Larsen. Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods. In T. Margaria, A. Philippou, and B. Steffen, editors, *Proc. 2nd Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2007)*, 2007.
- [9] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.
- [10] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

- [11] C.A.R. Hoare and J. He. *Unifying theories of programming*, volume 14. Prentice Hall, 1998.
- [12] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, MA, USA, 2003.
- [13] J. Holt and S. Perry. *SysML for Systems Engineering*. IET, 2008.
- [14] Jon Holt, Simon Perry, Mike Brownsword, Daniela Cancila, Stefan Hallerstede, and Finn Overgaard Hansen. Model-based requirements engineering for system of systems. In *Proceedings of the 7th Conference on Systems of Systems Engineering*, 2012.
- [15] Holt J., Perry S., and Brownsword M. *Model-Based Requirements Engineering*. IET, 2011.
- [16] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2007.
- [17] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153. Springer-Verlag, 1995.
- [18] Mark W. Maier. Architecting Principles for Systems-of-Systems. *Systems Engineering*, 1(4), 1998.
- [19] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, January 2007.
- [20] Bertrand Meyer. *Object-Oriented Software Construction (2nd ed.)*. Prentice-Hall, 1988.
- [21] Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. Model-based engineering of Systems of Systems. Technical report, COMPASS project, 2013. Available from <http://www.compass-research.eu/resources/sos.pdf>.
- [22] OASIS. Web Services Business Process Execution Language. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>, 2007.
- [23] Dependable Systems of Systems. Case Study Problem Analysis: The Travel Agent Problem. Technical Report IST-1999-11585, Dependable Systems of Systems, May 2011.
- [24] Object Management Group OMG. Unified Modelling Language infrastructure version 2.2. <http://www.omg.org/spec/UML/2.2/Infrastructure>, February 2009.
- [25] Object Management Group OMG. Systems Modelling Language version 1.2. <http://www.omg.org/spec/SysML/1.2>, June 2010.

- [26] International Council on Systems Engineering. Systems engineering handbook. a guide for system life cycle processes and activities. Technical report, INCOSE, January 2010.
- [27] Richard Payne, Jeremy Bryans, John Fitzgerald, and Steve Riddle. Interface Specification for System-of-Systems Architectures. In *Proceedings of the 7th Conference on Systems of Systems Engineering*, 2012.
- [28] Richard Payne and John Fitzgerald. Evaluation of Architectural Frameworks Supporting Contract-based Specification. Technical Report CS-TR-1233, School of Computing Science, Newcastle University, 2010.
- [29] Richard Payne and John Fitzgerald. Interface contracts for architectural specification and assessment: a sysml extension. In *Proceedings of the 2011 Workshop on Dependable Systems of Systems*, WDSoS '11, 2011.
- [30] The COMPASS project. CML Definition 1 (public). <http://www.compass-research.eu/deliverables.html>, 2012.
- [31] The COMPASS project. Report on modelling patterns for SoS architectures. <http://www.compass-research.eu/deliverables.html>, 2013.
- [32] W3C. Web Services Description Language v1.1. <http://www.w3.org/TR/wsdl>, 2001.
- [33] W3C. Web Services Choreography Description Language v1.0. <http://www.w3.org/TR/ws-cdl-10/>, 2005.
- [34] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: a Formal Modelling Language for Systems of Systems. In *Proceedings of the 7th International Conference on System of System Engineering*. IEEE, July 2012.
- [35] Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.
- [36] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.

A Travel Agent SoS SysML Model

The SoS model defined in SysML is presented in this appendix. The model is given as follows. In Section A.1, the structure of the case study is given and the interfaces are identified. The interfaces are defined in Sections A.2, A.3, A.4 and A.5. Finally, SoS-level scenarios are given in Section A.6.

The model provided does not include interface definitions for IF_HBS1 and IF_HBS2. This is due to the fact that these should be independently defined and the wrapper constituent systems must provide logic so to ensure compliance with the defined IF_CommonHotel interface. This is left for future work.

A.1 Travel Agent Structure

Figure 9: Block definition diagram depicting travel agent SoS structure.

Figure 10: Interface Connectivity View showing relationships of constituents.

A.2 IF_TripBooking Interface

Figure 11: IF_TripBooking Interface Definition View.

Figure 12: IF_TripBooking Interface Behaviour View.

Figure 13: IF_TripBooking Interface Behaviour View.

Figure 14: IF_TripBooking Interface Behaviour View.

Figure 15: IF_TripBooking Interface Behaviour View.

Figure 16: IF_TripBooking Protocol Definition View.

A.3 IF_CommonHotel Interface

Figure 17: IF_CommonHotel Interface Definition View.

Figure 18: IF_CommonHotel Protocol Definition View.

A.4 IF_CentralFlight Interface

Figure 19: IF_CentralFlight Interface Definition View.

Figure 20: IF_CentralFlight Interface Behaviour View.

Figure 21: IF_CentralFlight Interface Behaviour View.

Figure 22: IF_CentralFlight Protocol Definition View.

A.5 IF_CommonFlight Interface

Figure 23: IF_CommonFlight Interface Definition View.

Figure 24: IF_CommonFlight Interface Behaviour View.

Figure 25: IF_CommonFlight Protocol Definition View.

A.6 SoS-level Scenarios

Figure 26: SysML Sequence Behaviour Diagram showing complete SoS behaviour for booking and reserving flights and hotel.

Figure 27: SysML Sequence Behaviour Diagram showing complete SoS behaviour for timeout of reserved flights and hotel.

Figure 28: SysML Sequence Behaviour Diagram showing complete SoS behaviour for paying for reserved flights and hotel.

B Travel Agent SoS CML Model

The SoS model defined in CML is presented in this appendix. The model is given as follows. In Section B.1, the global types are defined and global channels are given in Section B.2. The interfaces are defined in Sections B.3, B.4, B.5 and B.6. Finally, the SoS composition is given in Section B.7. As in the SysML model, we do not model the individual hotel booking system interfaces.

B.1 Global Types

```
types
  FlightRequest :: startDate : Date
```

```

        endDate : Date
        fromLoc : Location
        toLoc: Location
        numPassengers : nat

FlightBookingId = token

Flight :: flightId : FlightId
        flightDate : Date
        frm : Location
        dest : Location
        seatsAvail : nat

Price = nat

ReserveId = token

FlightId = token

HotelId = token

Location = token

Date = token

FlightDatabase = map FlightId to Flight

BookingId = token

BookingRequest :: startDate : Date
                endDate : Date
                fromLoc : Location
                toLoc: Location
                numPassengers : nat
                flight : bool
                hotel : bool

BookingDetails :: startDate : Date
                endDate : Date
                fromLoc : Location
                toLoc: Location
                fId : FlightId
                hId : HotelId
                numPassengers : nat
                price : nat

HotelRequest :: startDate : Date
                endDate : Date
                loc : Location
                numRooms : nat

Hotel :: hotelId : HotelId
        loc : Location

HotelBookingId = token

```

B.2 Global Channels


```

channels

reqFlightAvail : FlightRequest
returnFlightAvail : set of Flight
reqFlightPrice : Flight * nat
returnFlightPrice : Price
reqReserveFlight : Flight * nat
returnReserveFlight : ReserveId
reqBookReserved : ReserveId
returnBookReserved : FlightBookingId
unreservedFlight : ReserveId
reqBookFlight : Flight * nat
returnBookFlight : FlightBookingId
reqGetFlights
returnGetFlights: FlightDatabase
reqProcessBooking : Flight * nat
returnProcessBooking: bool
reqGetQuote : BookingRequest
returnGetQuote : set of BookingDetails
reqPayQuote : BookingDetails
returnPayQuote : BookingId
reqReserve : ReserveId
returnReserve : BookingId
reqPayReserve : ReserveId
returnPayReserve : BookingId
reqHotelAvail : HotelRequest
returnHotelAvail : Hotel
reqHotelPrice : Hotel * nat
returnHotelPrice : Price
reqReserveHotel : Hotel * nat
returnReserveHotel : ReserveId
reqBookRooms : Hotel * nat
returnBookRooms : HotelBookingId
reqBookReservedHotel : ReserveId
returnBookReservedHotel : HotelBookingId
unreservedHotel

```

B.3 IF_TripBooking Interface

```

process IF_TripBooking =
begin

  state

    timeout : nat := 10

  operations

    public getQuote(req: BookingRequest) bds: set of BookingDetails
    post true

    public reserve(b : BookingDetails) r : ReserveId
    post true

    public payReserved(r : ReserveId) bId : BookingId
    post true

    public payQuote(b : BookingDetails) bId : BookingId

```

```

    post true

actions

    TB_Ready = reqGetQuote?req ->
        (dcl bds : set of BookingDetails @ bds :=
            getQuote(req);
            returnGetQuote!bds -> TB_Ready)
    []
    reqPayQuote?b ->
        (dcl bId : BookingId @ bId := payQuote(b);
            returnPayQuote!bId -> TB_Ready)
    []
    reqReserve?r ->
        (dcl bId : BookingId @ bId := reserve(r);
            returnReserve!bId -> TB_Reserved)

    TB_Reserved = (reqPayReserve?rId ->
        (dcl bId : BookingId @ bId := payReserved(rId);
            returnPayReserve!bId -> TB_Ready))
    [(timeout)>
        unreservedFlight?r -> TB_Ready

    @ TB_Ready
end

```

B.4 IF_CommonHotel Interface

```

process IF_CommonHotel =
begin

    state

        timeout : nat := 10
        private reservedHotels : set of ReserveId
        private bookedHotels : set of HotelBookingId

    operations

        public hotelAvail(req: HotelRequest) h: Hotel
        post forall f in set fls @ ((f.flightDate = req.startDate) and
            (f.dest = req.toLoc) and (f.seatsAvail >= req.numPassengers)
            )

        public hotelPrice(h: Hotel, n : nat) p : Price
        pre fl.seatsAvail >= n
        post p > 0

        public bookRooms(h: Hotel, n : nat) hbId : HotelBookingId
        pre r in set reservedFlights
        post fbId in set bookedFlights

        public reserveRooms(h: Hotel, n : nat) r : ReserveId
        pre fl.seatsAvail >= n
        post r in set reservedFlights

```

```

public releaseRooms(r : ReserveId) r : bool
pre r in set reservedFlights
post r not in set reservedFlights

public bookReserved(r : ReserveId) hbId : HotelBookingId
pre fl.seatsAvail >= n
post fbId in set bookedFlights

actions

  CH_Ready = reqHotelAvail?req ->
    (dcl h : Hotel @ h := hotelAvail(req);
     returnHotelAvail!h -> CH_Ready)
  []
  reqHotelPrice?h?num ->
    (dcl p : Price @ p := hotelPrice(h,num);
     returnHotelPrice!p -> CH_Ready)
  []
  reqReserveHotel?h?num ->
    (dcl r : ReserveId @ r := reserveRooms(h,num);
     returnReserveHotel!r -> CH_Reserved)
  []
  reqBookRooms?h?num ->
    (dcl hbId : HotelBookingId @ hbId := bookRooms(h,
      num);
     returnBookRooms!hbId -> CH_Ready)

  CH_Reserved = (reqBookReservedHotel?rId ->
    (dcl hbId : HotelBookingId @ hbId :=
      bookReserved(rId);
      returnBookReservedHotel!hbId -> CH_Ready))
  [(timeout)>
    unreservedHotel?r -> releaseRooms(r); CH_Ready

  @ CH_Ready
end

```

B.5 IF_CentralFlight Interface

```

process IF_CentralFlight =
begin

  state

    timeout : nat := 10
    private reservedFlights : set of ReserveId
    private bookedFlights : set of FlightBookingId

  operations

    public flightAvail(req: FlightRequest) fls: set of Flight
    post forall f in set fls @ ((f.flightDate = req.startDate) and
      (f.dest = req.toLoc) and (f.seatsAvail >= req.numPassengers)
      )

    public flightPrice(fl: Flight, n : nat) p : Price
    pre fl.seatsAvail >= n
    post p > 0

```

```

public reserveFlight(fl: Flight, n : nat) r : ReserveId
pre fl.seatsAvail >= n
post r in set reservedFlights

public bookReserved(r : ReserveId) fbId : FlightBookingId
pre r in set reservedFlights
post fbId in set bookedFlights

public releaseFlight(r : ReserveId)
pre r in set reservedFlights
post r not in set reservedFlights

public bookFlight(fl: Flight, n : nat) fbId : FlightBookingId
pre fl.seatsAvail >= n
post fbId in set bookedFlights

actions

  CF_Ready = reqFlightAvail?flreq ->
    (dcl s : set of Flight @ s := flightAvail(flreq);
     returnFlightAvail!s -> CF_Ready)
  []
  reqFlightPrice?fl?num ->
    (dcl p : Price @ p := flightPrice(fl,num);
     returnFlightPrice!p -> CF_Ready)
  []
  reqReserveFlight?fl?num ->
    (dcl r : ReserveId @ r := reserveFlight(fl,num);
     returnReserveFlight!r -> CF_Reserved)
  []
  reqBookFlight?fl?num ->
    (dcl fbId : FlightBookingId @ fbId := bookFlight(
      fl,num);
     returnBookFlight!fbId -> CF_Ready)

  CF_Reserved = (reqBookReserved?rId ->
    (dcl fbId : FlightBookingId @ fbId :=
      bookReserved(rId);
     returnBookReserved!fbId -> CF_Ready))
  [(timeout)>
    unreservedFlight?r -> releaseFlight(r);
    CF_Ready

  @ CF_Ready
end

```

B.6 IF_CommonFlight Interface

```

process IF_CommonFlight =
begin

  operations

  public getFlights() fDb : FlightDatabase
  post true

  public processBooking(fl: Flight, n : nat) b : bool

```

```

pre fl.seatsAvail >= n
post true

actions

  CF_Ready = reqGetFlights ->
    (dcl f : FlightDatabase @ f := getFlights();
     returnGetFlights!f -> CF_Ready)
  []
  reqProcessBooking?fl?num ->
    (dcl b : bool @ b := processBooking(fl,num);
     returnProcessBooking!b -> CF_Ready)

  @ CF_Ready
end

```

B.7 Complete SoS Process

```

process SoS = IF_TripBooking ||| IF_CommonHotel |||
  IF_CentralFlight ||| IF_CommonFlight

```