

COMPUTING SCIENCE

Interface Specification for System-of-Systems Architectures

Richard Payne, Jeremy Bryans, John Fitzgerald and Steve Riddle

TECHNICAL REPORT SERIES

No. CS-TR-1323

May 2012

Interface Specification for System-of-Systems Architectures

R. Payne, J. Bryans, J. Fitzgerald, S. Riddle

Abstract

Establishing that a system-of-systems (SoS) architecture respects global SoS-level properties is complex. Recording explicit technical interfaces at the boundaries of constituent systems would facilitate this, but support for the description of such interfaces is limited in current widely-used architectural notations. This paper identifies research challenges that arise from using the combination of SysML and the formal notation VDM to describe the interface specifications recorded at the boundaries of the constituent systems. The approach is illustrated with a case study based on an emergency services SoS.

Bibliographical details

PAYNE, R., BRYANS, J., FITZGERALD, J., RIDDLE, S.

Interface Specification for System-of-Systems Architectures
[By] R. Payne, J. Bryans, J. Fitzgerald, S. Riddle

Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1335)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1335

Abstract

Establishing that a system-of-systems (SoS) architecture respects global SoS-level properties is complex. Recording explicit technical interfaces at the boundaries of constituent systems would facilitate this, but support for the description of such interfaces is limited in current widely-used architectural notations. This paper identifies research challenges that arise from using the combination of SysML and the formal notation VDM to describe the interface specifications recorded at the boundaries of the constituent systems. The approach is illustrated with a case study based on an emergency services SoS.

About the authors

Richard received his BSc (Hons) in Computing Science from Newcastle University in 2005. He obtained his PhD in 2012 at Newcastle University under the supervision of Dr. John Fitzgerald, as part of the DIRC project, 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-system architectural models. Richard is now working on the COMPASS project, on the use of model-based techniques for developing and maintaining systems-of-systems.

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.

John Fitzgerald is Director of the Centre for Software Reliability (CSR) at Newcastle. He is a specialist in the engineering of resilient computing systems, particularly in rigorous analysis and design tools. In his research, he develops model-based methods and tools to help in the design of particularly challenging types of product, especially systems that require collaboration between engineering teams of differing backgrounds and disciplines. For example, he currently leads the international COMPASS project, which is developing technology for engineering complex "Systems-of-Systems" that are built from pre-existing systems that might never have been designed with collaboration in mind. On a different scale, he leads Newcastle's research into co-modelling and co-simulation in the design of fault-tolerant embedded systems (in the DESTTECS project and in our EPSRC platform grant on Trustworthy Ambient Systems). John is probably most closely associated with the Vienna Development Method (VDM) which has been developed from its logical foundations to a commercial tool-supported method, with industry applications in areas as diverse as options trading and firmware design. He recently led work in the Deploy project on achieving and demonstrating dependability through the deployment of formal methods in four industry sectors. His project on the use of formal models to support collaborative modelling and simulation in the design of embedded systems (DESTTECS), started in January 2010. John studied formal proof (PhD, Manchester Univ.), before joining Newcastle, where he worked on formal design techniques for avionic systems with British Aerospace in the 1990s. He went on to study the potential for industrial application of formal modelling (specifically VDM) as a SERC Fellow and later as a Lecturer at Newcastle. He returned to the University in 2003, having established the design and validation team at Transitive, a successful SME in the embedded processor market. John is Chairman of FME, the main European body bringing together researchers and practitioners in rigorous methods of systems development. He is a Fellow of the BCS, and a member of the EPSRC College. He is a member of the ACM and IEEE.

Dr Steve Riddle is a lecturer in Computing Science. His research contributions have included novel safety analysis techniques (INCO-COPERNICUS ISAT), component contracts and protective wrapper architectures (EPSRC project DOTS) and resilience (FP6 NoE ReSIST). He is a Theme Lead in the SSEI project and leads work in dependable dynamic reconfiguration in that project. Beside these projects his research interests include requirements volatility, evidence-based argumentation and risk management.

Suggested keywords

INTERFACES
CONTRACTS
SYSTEMS-OF-SYSTEMS
VDM, ARCHITECTURE
SYSML

Interface Specification for System-of-Systems Architectures

Richard Payne, Jeremy Bryans, John Fitzgerald, Steve Riddle
School of Computing Science, Newcastle University, UK

May 2012

Abstract

Establishing that a system-of-systems (SoS) architecture respects global SoS-level properties is complex. Recording explicit technical interfaces at the boundaries of constituent systems would facilitate this, but support for the description of such interfaces is limited in current widely-used architectural notations. This paper identifies research challenges that arise from using the combination of SysML and the formal notation VDM to describe the interface specifications recorded at the boundaries of the constituent systems. The approach is illustrated with a case study based on an emergency services SoS.

1 Introduction

The characteristics of systems-of-systems (SoSs) identified by Maier [14] make SoS engineering a significant challenge. At the same time, the complexity of SoSs and the reliance placed on them is increasing. Methods and tools for SoS engineering are therefore required in order to permit the SoS engineer to validate global SoS properties during design and evolution. An important area of potential benefit is the description and analysis of SoS architectures.

Although SoS engineering offers unique challenges, we believe it to be beneficial to adapt current methods and tools as starting points [6] rather than supplanting current best practice. In our view, a promising method for SoS architectural definition lies in interface specification, because it allows the internal definition of constituent systems to change, as long as it continues to respect the contractual interface specification. In developing techniques for modelling SoS architectures we therefore start from the established notation SysML [18]. While designed for systems engineering, several features of SysML mean that it can be used for description of SoS architectures. However, like many architectural description languages, support for the formal specification of the interfaces between the constituent systems is limited, and this in turn limits the extent to which automated or partially automated tools can be used to analyse semantically significant properties of models.

A possible approach to improving the level of formality is to use an existing formal specification language in combination with SysML, allowing tools and methods developed for the formalism to be used alongside those of SysML. Formal specification languages have a mathematically well-founded and precisely defined semantics. They have associated techniques that allow desirable properties of a system to be specified and demonstrated to a high degree of rigour. A suitable language is VDM [9], a formal

model-based specification language. The analysis techniques for VDM include static analysis by syntax-checking, type-checking and proof, and dynamic analysis through testing and simulation. These analyses can increase the confidence that a SoS engineer has on the correctness of global SoS properties.

This paper identifies research challenges that arise from using SysML in combination with VDM to describe the interfaces of the constituent systems of SoSs. We illustrate these using a case study from the emergency response domain. We begin by summarising the current state of the art in interface contract specification in Section 2, then introduce a case study based on an emergency response scenario in Section 3 with the SoS architecture defined in SysML, identifying some of the interfaces in the study and the specification of interfaces in VDM. Section 4 considers the design and analysis of these interfaces using VDM. Section 5 concludes.

2 Interface Specification for Architectural Modelling

2.1 State of the Art in Interface Specification in Architectural Modelling

The state of the art in interface specification in architectural description notations is poor [21]. The most widely used notations (UML [17] and SysML [18]) allow basic signatures to be defined and pre- and postconditions to be specified textually, but these are rarely used. In AADL [8] 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 [13] and Wright [2] allow software components to have ports defined, and (in the case of Wright) have their message exchange protocols defined. However, these notations do not include the ability to specify other details such as operation pre/postconditions, and they do not contain architectural abstractions suitable for the definition of SoSs.

2.2 Interface Contracts and Design by Contract

Interface contracts are descriptions of the constituent systems of a SoS described *contractually* in terms of their expectations and the obligations placed on their behaviour. They have much in common with the idea of Design by Contract, a software engineering technique introduced by Meyer [15] in which contracts make explicit the relationships between systems in terms of preconditions and postconditions on operations and invariants on states. In Meyer's approach, contracts mainly specify functionality. The interaction between operations can be described using notations such as UML sequence diagrams or in process algebraic notation such as CSP [11] or CCS [16].

The use of contracts in service selection and subscription is an active research field in service-oriented computing, in particular the use of contracts for the specification of non-functional properties. Beugnard et al. [3] expand the notion of a contract to architectures in which components provide services. A four-level structure for contracts is proposed, adding 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.

2.3 Potential Benefits of Interface Contracts

The incorporation of interface contracts in architectural specifications may provide two main benefits:

- Interface contracts defined for the constituent systems of a SoS architectural design permit the analysis of SoS-level properties. These analyses give SoS designers the ability to experiment with consequences of different architectural designs.
- SoS designers can define the expected interfaces of the constituent systems, and these definitions may be provided to the system developers. This provides greater confidence to SoS designers that constituent systems will adhere to the expected properties on interfaces.

2.4 An Approach to Interface Contract Specification

In this paper we consider interface specification defined contractually for SoS architectural modelling. Based on the existing state of the art in architectural modelling, identified in Section 2.1, current notations are limited for specifying interface contracts. As highlighted in Section 1, however, we believe it to be beneficial to adapt existing notations from systems engineering practice as starting points. As such, we propose the use of the SysML, widely-used in industry, to define SoS architectures, along with the use of VDM to formally define interface contracts identified in SysML models. The contribution of the paper is to identify the research challenges that arise from using the combination of SysML with VDM to describe the interfaces of existing constituent systems of SoSs.

3 LESLP Case Study

In order to explore the consequences of formal interface specification in a contractual style, we use a case study based on the system formed by emergency services (fire, police, ambulance etc.) in response to a major incident. We refer to this system as the Major Incident Response (MIR). The MIR is a SoS in Maier's terms [14] and may be considered an *acknowledged* SoS [7]. The emergency services are operationally and managerially independent. Each service is itself geographically distributed and may evolve, for example as personnel come on and off duty during the course of a long-running incident. Emergent behaviour is also present – the comprehensive approach to management of the incident relies on voluntary and collaborative interaction. We give more details on the case study in Section 3.1, and an architectural description in Section 3.2. Section 3.3 supplements this description with a formal definition of the interfaces.

3.1 Informal Description of the Case Study

We base the study on the procedures for the coordination of the MIR in London, as outlined in the Major Incident Procedure Manual [19] published by the London Emergency Services Liaison Panel (LESLP). This documents the process for identifying a major incident, the initial information to be passed to the appropriate services and the roles and responsibilities of the service members at the scene.

The response to all major incidents follows a broadly similar structure. The members from each service attending the scene form Bronze command. For more severe incidents, a Silver command will be formed containing representatives of all the involved services. For long-running incidents, a Gold command may be formed at a geographically distant point. The Bronze, Silver, Gold hierarchy corresponds to the operational, tactical and strategic levels of command.

We pay particular attention to the rules outlined in [19] for communication of casualty information with the media, and the requirements these place on the interfaces between the emergency services. In the early stages of a major incident, confusion can arise if the media aggregate casualty figures from various sources. This can lead to “double-counting” and overestimation of the severity of incident. To avoid this, all casualty details must be given to Gold command, which is then responsible for coming to a more reliable estimate and communicating this to the media.

In [5] the case study was explored using the Event-B [1] formalism, but the model developed there does not provide an accessible representation of the SoS architecture, and does not consider the interface specification between the constituent systems. In this paper we focus on these aspects.

3.2 Architectural Description of LESLP

SysML [18] is a profile for UML 2.0, developed for system engineering, but also supporting the modelling of SoS architecture. It enjoys wide industrial support and a sound tool base. SysML provides several diagram types, with a “precise natural language” semantics, to support the description of the SoS structure, behaviour and requirements.

The MIR structure of the case study is given in Figure 1 as a SysML Block Definition Diagram (BDD). The MIR contains up to three emergency services and these are the constituent systems of the MIR SoS. The emergency services (ES) may be a police force, a fire brigade or ambulance service. All ES contain one or more person and each person has a role (Bronze, Silver or Gold).

We consider one requirement of the MIR SoS, that only accurate casualty information should be released to the media, and show in detail how the constituent systems communicate¹. These communications between instances of the constituent systems with a given role are described in an Internal Block Diagram (IBD) given in Figure 2. The IBD details the provided and required interfaces of the systems of the MIR SoS and depends upon the roles undertaken by the constituents. For example, in the IBD of Figure 2, an interface *info_to_silver* exists between Officers with a Bronze role and their respective emergency service Officers with Silver role.

The SysML interface definitions in Figure 3 relate to points of interaction for those operations made public by the relevant constituent systems. The operations are defined in terms of operation signatures detailing data input to and output from an operation call. For example, the *info_to_silver* interface, relating to the transferring of casualty information from Bronze officers to Silver Ambulance officers, consists of the operation *verifyCasualtyDetails(CasualtyDetails)* which requires some unverified casualty information of the *CasualtyDetails* data type (defined elsewhere in the model).

The SysML specification allows pre- and postconditions to be specified for operations, however this is optional and no analyses are available to ensure their correctness. SysML also omits the Protocol State Machine of UML 2.0 which dictates the response of an interface to specified sequences of events, constraining the order of operations. It

¹We consider only the communications and interactions that are necessary to meet this requirement.

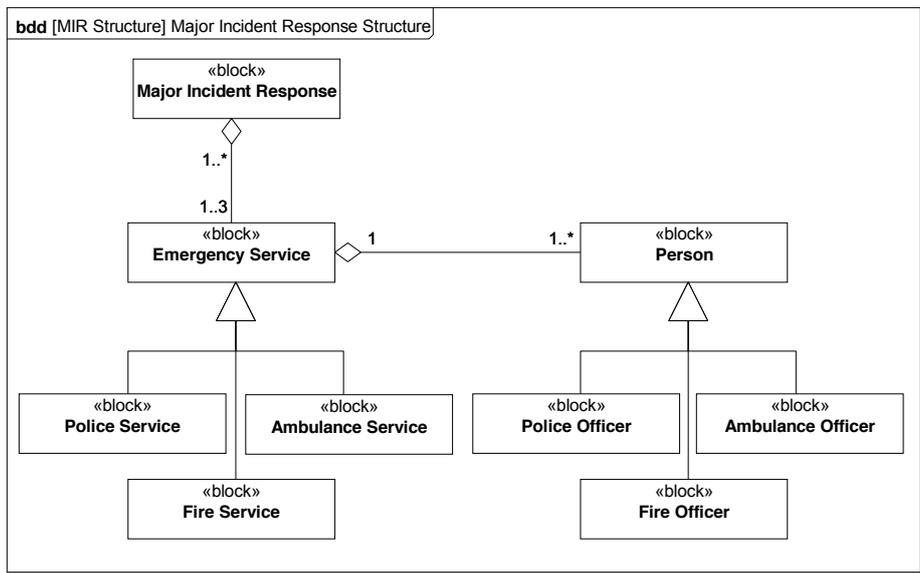


Figure 1: Block definition diagram depicting Major Incident Response structure.

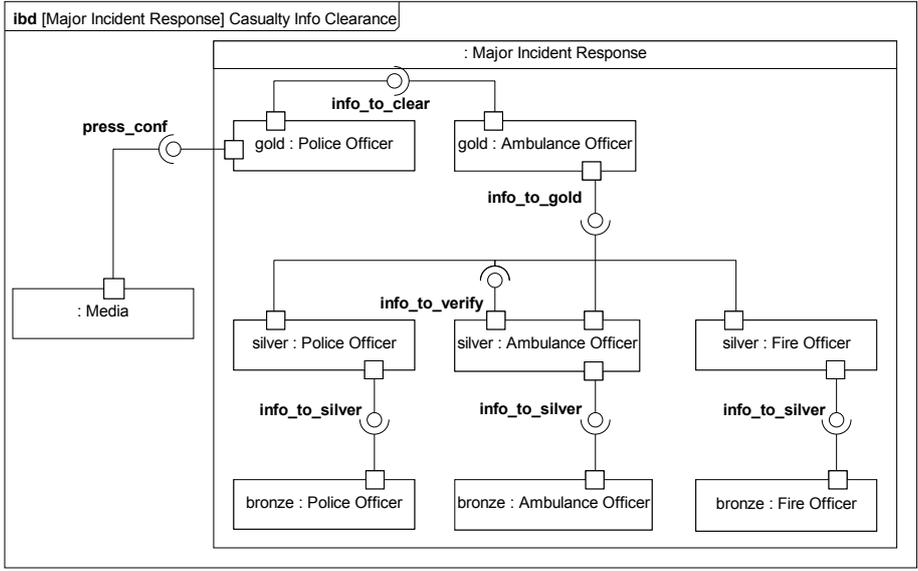


Figure 2: Internal Block Diagram showing relationships of response constituents when releasing casualty figures.

is our opinion that this construct would add additional rigour to an interface specification and increase the range of analyses available.

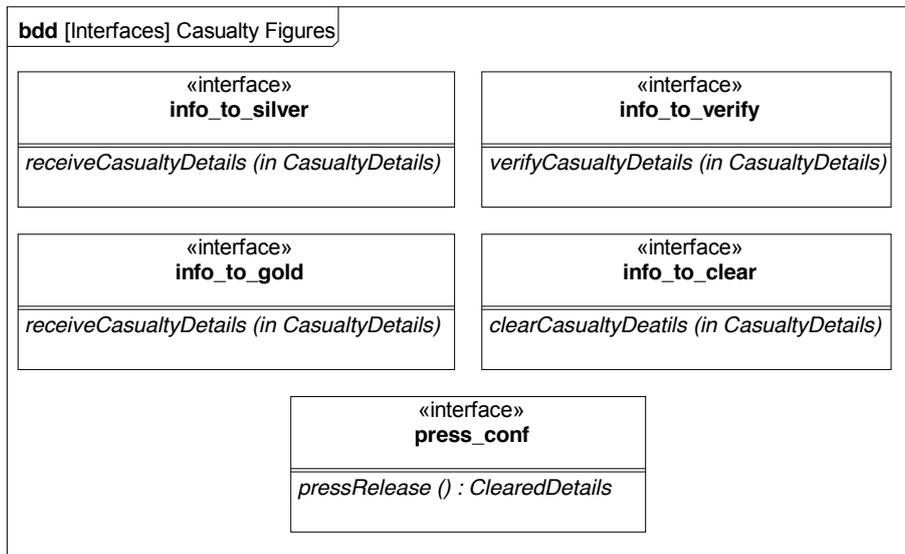


Figure 3: Block Definition Diagram showing interface definitions when releasing casualty figures.

3.3 Formal Definition of LESLP Interface Contracts

Given a SysML architectural specification, we strengthen the interface specifications of the constituent systems as described above. Using a formal model-based specification language allows increased confidence using a range of analysis techniques. In this paper, VDM is used to give formal definitions of the architectural interfaces. VDM is a model-oriented notation supporting descriptions of data and functionality. VDM has industry-strength² and open-source³ tool support.

The SysML interfaces in Figure 3 may be modelled as VDM classes. VDM supports inheritance, so we define the operations of interfaces with pre- and postconditions, and provide no operation body at the interface. The operation definition, therefore, does not state *what* is done, the implementation of the algorithm must be provided by the constituent systems implementing the interface. This is denoted using the *is subclass responsibility* key phrase in the operation body. The system designer must give a specification for each constituent system which provides the interface, and ensure that the implementation of the constituent system contains the interface operations, and that they are consistent with the interface specification.

Figures 4a) and 4b) define interface definitions of the *info_to_clear* and *press_conf* interfaces initially given in Figure 3. The *operations* keyword in each class denotes the available operations of the interfaces with signatures corresponding to those defined in Figure 3. Each operation is strengthened using pre- and postconditions. The *clearCasualtyDetails* operation of the *info_to_clear* interface, shown in Figure 4a), requires a parameter *cd*, of type *CasualtyDetails*. The operation does not return any result. As mentioned above, no operation body is given. The operation precondition, denoted by

²VDMTools: <http://www.vdmtools.jp/en>

³Overture: <http://www.overturetool.org>

the keyword *pre*, states that there must be 0 or greater reported casualties (*cd.number* ≥ 0), they must have been verified (*cd.verified*), and are not already cleared for release (*not cd.cleared*). No postcondition is provided.

The *press_conf* interface in Figure 4b) contains one operation, *pressRelease*. The operation has no parameters and returns a result of type *ClearedDetails*. This interface operation also has no body. The operation has no precondition, but has a postcondition, which is given after the *post* keyword. We may refer to the return variable using the *RESULT* keyword. The postcondition requires that the result has been verified and cleared (*RESULT.verified and RESULT.cleared*).

```
class info_to_clear

  operations
  public clearCasualtyDetails: CasualtyDetails ==> ()
  clearCasualtyDetails(cd) ==
  is subclass responsibility
  pre cd.number >= 0 and cd.verified and
  not cd.cleared

end info_to_clear
```

(a) *info_to_clear* interface.

```
class press_conf

  operations
  public pressRelease: () ==> ClearedDetails
  pressRelease () ==
  is subclass responsibility
  post RESULT.verified and RESULT.cleared;

end press_conf
```

(b) *press_conf* interface.

Figure 4: VDM representations of interface definitions.

Given these interface specifications, VDM classes are defined which implement the various interface definitions to correspond with the IBD in Figure 2. These classes are implemented using the *is subclass of* key phrase followed by the interface class name. The classes (for example Gold Police) also implement the *Police* abstract class detailing emergency service-specific variables.

The Gold Police system is defined as a VDM class below in Figures 5, 6 and 7. Figure 5 shows the Gold Police class which implements the *info_to_clear* and *press_conf* interfaces, defined on the first two lines. The class has a single instance variable, *cas_cleared*, a set of type *CasualtyDetails*, initially set to the empty set $\{\}$.

In Figure 6, the Gold Police class implements the *clearCasualtyDetails* operation, as defined in the *info_to_clear* interface. The operation strengthens the precondition to ensure that the police officer has the rank Gold (*role = <Gold>*) and the remainder of the precondition is unchanged. The operation postcondition ensures that the number of *CasualtyDetails* items in the *cas_cleared* instance variable does not decrease (*card cas_cleared \geq card cas_cleared⁴*). The postcondition also states that, for all *CasualtyDetails* items in the *cas_cleared* set which have the same location as the parameter

⁴In VDM postconditions, a variable name succeeded by the \sim symbol refers to the initial value of that variable.

```

class Gold_Police is subclass of Police,
    info_to_clear, press_conf

instance variables
private cas_cleared : set of CasualtyDetails := {};
...
end Gold_Police

```

Figure 5: VDM class representing *Gold Police* constituent system: preamble and instance variables.

cd, those details should be verified and cleared (*forall c in set cas_cleared & c.loc = cd.loc => c.verified and c.cleared*).

```

class Gold_Police
...
operations
public clearCasualtyDetails : CasualtyDetails ==> ()
clearCasualtyDetails(cd) ==
(
  if not exists c in set cas_cleared & c.loc = cd.loc
  then cas_cleared := cas_cleared union
    {mk_CasualtyDetails(cd.number, true,
                       true, cd.type, cd.loc)}
)
pre role = <Gold> and cd.number >= 0 and
  cd.verified and not cd.cleared
post card cas_cleared >= card cas_cleared~ and
  forall c in set cas_cleared & c.loc = cd.loc =>
    c.verified and c.cleared;
...
end Gold_Police

```

Figure 6: VDM class representing *Gold Police* constituent system: *clearCasualtyDetails* operation.

The operation body of the *clearCasualtyDetails* operation, given after the operation signature, has an *if* statement as the main structure. If there does not exist any casualty details in the *cas_cleared* set with the same location as the parameter *cd* (*if not exists c in set cas_cleared & c.loc = cd.loc then*), then a new *CasualtyDetails* item is added to the *cas_cleared* set with the *verified* and *cleared* fields set to *true*, all other fields are given as those supplied by the parameter, *cd* (*cas_cleared := cas_cleared union {mk_CasualtyDetails(cd.number, true, true, cd.type, cd.loc)}*).

The *Gold Police* class also implements the *press_conf* interface and so, in Figure 7, the *pressRelease* operation is given. The *pressRelease* operation in Figure 7 has a strengthened precondition, as with the *clearCasualtyDetails* operation, to ensure that police officers carrying out the operation have the rank *<Gold>*. The postcondition is also strengthened to ensure that the number of casualty details released is less than, or equal to, the number of cleared details the *Gold Police* know about. The operation uses a private function, *totalCleared*, to calculate the total cleared figure (*RESULT.number*

\leq $totalCleared(cas_cleared)$). This is an important property of the casualty clearance scenario – that the Major Incident Response command do not release casualty figures exceeding the number of casualties that have been verified and cleared for release. The Gold Police are able to use their discretion in releasing a lower figure than is known. The *pressRelease* operation body passes the *cas_cleared* instance variable to a private *clearForPress* function, the result of which is returned (**return** *clearForPress(cas_cleared)*). The private *clearForPress* and *totalCleared* functions are given in Figure 7. The *clearForPress* operation body is undefined using the *is not yet specified* key phrase – a policy decision for the Gold Police, not given here. Finally, the *totalCleared* function is a simple recursive function to count the number of casualties given a *CasualtyDetails* set.

```

class Gold_Police
...
operations
public pressRelease : () ==> ClearedDetails
pressRelease () ==
(
  return clearForPress(cas_cleared);
)
pre role = <Gold>
post RESULT.cleared and RESULT.verified and
  RESULT.number <= totalCleared(cas_cleared);

functions
private clearForPress : set of CasualtyDetails
  ==> ClearedDetails
clearForPress(cds) ==
  is not yet specified;

private totalCleared : set of CasualtyDetails -> nat
totalCleared(cds) ==
  cases cds:
    {} -> 0,
    others -> let cd in set cds in
      cd.number + totalCleared(cds\{cd})
  end;

end Gold_Police

```

Figure 7: VDM class representing *Gold Police* constituent system: *pressRelease* operation and auxiliary private functions.

The remainder of the VDM model of the Major Incident Response is given in Appendix A

4 Analysis of Interface Contracts

The purpose of a formal analysis is to confirm or refute specified properties that are required of the model under consideration. The analysis techniques available for VDM include static and dynamic techniques, with varying degrees of machine support. VDM's particular history of use in industry settings requiring extensive test-based validation

of models and model-based testing of implementations mean that its tool sets have highly developed interpreters allowing rapid testing of models on high volumes of test cases [12]. Simulation is closely linked to testing. Rather than executing a single well-defined test, the model execution is driven by a *scenario* containing multiple decision points that may be resolved by a user interacting with the model. This allows those not experienced in the notation or involved in the development to gain familiarity with the formal model, and provides a valuable way of exposing application domain experts to the model at an early stage.

VDM has a well-defined formal semantics, and therefore VDM models are amenable to logical proof [4]. Proof obligations arise naturally within a model, for example the obligation to prove that the specification of each operation is *satisfiable*, i.e., for all valid pre-states and inputs there is always a state of the model that satisfies the postcondition of the operation. Overture generates proof obligations automatically, and manages their manual “sign off” by the user, but currently little help is given to the user in automatically discharging them.

Testing, simulation and proof both have a contribution to make in increasing our assurance of the design of the MIR SoS. For example, both proof and unit testing of the operations would be valuable to ensure that explicit definitions for operations meet the implicit (pre- and postcondition) specifications. Proof is an expensive technique, and best applied only to a small number of key properties, but it provides a higher degree of confidence in a system. The MIR SoS was designed to ensure that only verified casualty figures were released to the media. Some properties relating to this purpose are suggested in [20], for example that *only Police Gold is authorised to release casualty figures*, or that *Unverified casualty figures must never be released to the general public*. Demonstrating properties such as these would be an appropriate application of proof technology.

5 Conclusion

From the work documented in this paper, we conclude certain requirements on the notation used for the specification of interfaces between constituent systems. Significant requirements include *the necessity of using a formal notation which includes architectural abstractions* and *the ability to describe the accepted orderings of events at the interface*. We also observe *the necessity to provide strong ties to an accepted industrial strength architectural description language* and *the ability to deal with different levels of abstraction*.

Whilst SysML enables SoS engineers to model complex SoS architectures, the facilities for interface definition are less satisfactory and little analysis is possible. Using a formal notation enables these analyses. However, existing formal notations with architectural abstractions are poor. We propose the use of formal specification notations, such as VDM, to define interfaces and concrete system specifications and to reason about their properties. In the paper, we demonstrate the use of VDM by defining interfaces corresponding to the SysML architectural model, and further specify systems implementing the interfaces. Whilst not performed in this paper, VDM allows strong static analysis support and dynamic support in the form of simulation, testing and proof obligation generation, in particular allowing the specification of interfaces at different levels of abstraction.

Current VDM tools lack model checking and proof support. Further, although it supports data-based specification of functionality, VDM does not contain abstractions

to support description of event *orderings* at interfaces. Existing notations which could fill these gaps in analyses include the family of process algebras, e.g. CSP. This would allow the definition of protocols on interfaces and provide dynamic analysis through model checking, which could be used to avoid protocol mismatch [10]. An optimal approach however, would be to use a notation that provides data-based modelling (as in VDM) *and* event ordering (such as CSP) and which also contains the abstractions necessary to model SoS architectures and state and reason over global SoS properties.

The development of such a notation is the goal of our current work in the COMPASS project⁵. This project aims to improve the state of the art in SoS engineering by provision of modelling tools and analysis techniques based on an underlying modelling language (the COMPASS Modelling Language, CML). CML provides VDM-style data modelling and CSP-style event ordering as outlined above, for representing SoS architectures and interface contracts. This language will have a formal semantic definition to support description of behaviour and composition of subsystem properties, based on Hoare and He's Unifying Theories of Programming (UTP) [?] and will integrate with SysML to support modelling using either textual CML, graphical SysML or a combination of the two.

Acknowledgments

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).

References

- [1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
- [2] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997.
- [3] Antoine Beugnard, Jean-Marc Jézéquel, Noel Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.
- [4] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.
- [5] Jeremy W. Bryans, John S. Fitzgerald, and Tom McCutcheon. Refinement-based techniques in the analysis of information flow policies for dynamic virtual organisations. In *Adaptation and Value Creating Collaborative Networks - 12th IFIP WG 5.5 Working Conference on Virtual Enterprises, PRO-VE 2011. Proceedings*, number 362 in IFIP Advances in Information and Communication Technology, pages 314–321, 2011.
- [6] Radu Calinescu and Marta Kwiatkowska. Software Engineering Techniques for the Development of Systems of Systems. In *Foundations of Computer Software*.

⁵www.compass-research.eu

- Future Trends and Techniques for Development*, volume 6028 of *Lecture Notes in Computer Science*, pages 59–82. Springer Berlin/Heidelberg, 2010.
- [7] 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.
 - [8] 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.
 - [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] Carl Gamble. *Design Time Detection Of Architectural Mismatches In Service Oriented Architectures*. PhD thesis, School of Computing Science, Newcastle University, UK, 2011.
 - [11] Tony Hoare. *Communication Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.
 - [12] T. Kurita and Y. Nakatsugawa. The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, 3(2-3), October 2009.
 - [13] 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.
 - [14] Mark W. Maier. Architecting Principles for Systems-of-Systems. *Systems Engineering*, 1(4), 1998.
 - [15] Bertrand Meyer. *Object-Oriented Software Construction (2nd ed.)*. Prentice-Hall, 1988.
 - [16] R. Milner. *Communication and Concurrency*. Prentice Hall International (U.K.) Limited, 1989.
 - [17] Object Management Group OMG. Unified Modelling Language infrastructure version 2.2. <http://www.omg.org/spec/UML/2.2/Infrastructure>, February 2009.
 - [18] Object Management Group OMG. Systems Modelling Language version 1.2. <http://www.omg.org/spec/SysML/1.2>, June 2010.
 - [19] London Emergency Services Liaison Panel. *Major Incident Procedure Manual*, seven edition, 2007. Available at www.met.police.uk/leslp.
 - [20] Richard J. Payne and Jeremy Bryans. Modelling the Major Incident Procedure Manual: A System of Systems Case Study. Technical Report CS-TR-1320, School of Computing Science, Newcastle University, March 2012.
 - [21] Richard J. Payne and John S. Fitzgerald. Evaluation of Architectural Frameworks Supporting Contract-based Specification. Technical Report CS-TR-1233, School of Computing Science, Newcastle University, December 2010.

A VDM Models of LESLP Case Study

A.1 Interface Definitions

A.1.1 Interface Datatypes

```
class IF_Types

  -- Basic Type definitions to be used in the Major Incident Model --
  types

  public CasualtyDetails :: number : nat
                          verified : bool
                          cleared : bool
                          type : token
                          loc : Location;

  public ClearedDetails :: number : nat
                          verified : bool
                          cleared : bool
                          type : set of token;

  public Location = token;
end IF_Types
```

A.1.2 info_to_silver

```
class info_to_silver

  operations
  -- Receive Collected Casualty Details operation
  --
  -- parameters:   cd : CasualtyDetails
  -- precondition: ensures the supplied cd parameter has not been
  --               verified or cleared
  public receiveCollectedCasualtyDetails : IF_Types`CasualtyDetails ==> ()
  receiveCollectedCasualtyDetails(cd) ==
  is subclass responsibility
  pre not(cd.verified) and not(cd.cleared) and cd.number >= 0;

end info_to_silver
```

A.1.3 info_to_verify

```
class info_to_verify

  operations

  -- Verify Casualty Details operation
  --
  -- parameters:   cd : CasualtyDetails
  -- precondition: ensures the supplied cd parameter has not been
```

```

--          verified or cleared, and the details have been
--          collected
public verifyCasualtyDetails : IF_Types\CasualtyDetails ==> ()
verifyCasualtyDetails(cd) ==
  is subclass responsibility
  pre not (cd.verified) and not (cd.cleared) and cd.number >= 0
end info_to_verify

```

A.1.4 info_to_gold

```

class info_to_gold

  operations
  -- Receive Casualty Details operation.
  --
  -- parameters:  cd : CasualtyDetails
  -- precondition: the casualty details have been verified, not
  --               cleared and greater than or equal to 0
  public receiveVerifiedCasualtyDetails : IF_Types\CasualtyDetails ==> ()
receiveVerifiedCasualtyDetails(cd) ==
  is subclass responsibility
  pre cd.number >= 0 and cd.verified and not cd.cleared
end info_to_gold

```

A.1.5 info_to_clear

```

class info_to_clear

  operations
  -- Receive Casualty Details operation.
  --
  -- parameters:  cd : CasualtyDetails
  -- precondition: the casualty details have been verified, not
  --               cleared and greater than or equal to 0
  public clearCasualtyDetails : IF_Types\CasualtyDetails ==> ()
clearCasualtyDetails(cd) ==
  is subclass responsibility
  pre cd.number >= 0 and cd.verified and not cd.cleared
end info_to_clear

```

A.1.6 press_conf

```

class press_conf

  operations
  -- Press Release operation
  --

```

```

-- returns:          res : ClearedDetails
-- postcondition: Cleared details are marked as cleared and verified
public pressRelease : () ==> IF_Types\ClearedDetails
pressRelease () ==
  is subclass responsibility
post RESULT.cleared and RESULT.verified;
end press_conf

```

A.1.7 order_to_collect_info

```

class order_to_collect_info

  operations
  -- Collect Casualty Details operation
  --
  -- parameters:  loc: Location
  public collectCasualtyDetails: IF_Types`Location ==> ()
  collectCasualtyDetails(loc) ==
    is subclass responsibility
end order_to_collect_info

```

A.2 Constituent System Types

A.2.1 Police

```

class Police is subclass of Person

  types

  public PoliceId = token;
  public PoliceRank = <Constable> | <Sergeant> | <Inspector> |
                    <Commander> | <Commissioner> | <UnAssigned>;

  instance variables

  protected id : PoliceId;
  protected rank : PoliceRank;

  operations
  -- Constructor --
  -----
  public Police : PoliceId * IF_Types`Location ==> Police
  Police(pid, l) ==
  (
    id := pid;
    rank := <UnAssigned>;
    role := <UnAssigned>;
    location := l;
  );

  public getId : () ==> PoliceId
  getId() == return id;

```

```
end Police
```

A.2.2 Ambulance

```
class Ambulance is subclass of Person

types

public AmbulanceId = token;
public AmbulanceRank = <Paramedic> | <AmbulanceManager> |
                       <UnAssigned>;

instance variables

protected id : AmbulanceId;
protected rank : AmbulanceRank;

operations
-- Constructor --
-----
public Ambulance : AmbulanceId * IF_Types`Location ==> Ambulance
Ambulance(aid, l) ==
(
  id := aid;
  rank := <UnAssigned>;
  role := <UnAssigned>;
  location := l;
);

public getId : () ==> AmbulanceId
getId() == return id;
end Ambulance
```

A.2.3 Fire

```
class Fire is subclass of Person

types

public FireId = token;
public FireRank = <Firefighter> | <CrewManager> | <Commissioner> |
                 <UnAssigned>;

instance variables

protected id : FireId;
protected rank : FireRank;

operations
-- Constructor --
-----
public Fire : FireId * IF_Types`Location ==> Fire
Fire(fid, l) ==
```

```

(
  id := fid;
  rank := <UnAssigned>;
  role := <UnAssigned>;
  location := l;
);

public getId : () ==> FireId
getId() == return id;

end Fire

```

A.3 Constituent System Definitions

A.3.1 Gold Police

```

class Gold_Police is subclass of Police, press_conf, info_to_clear

instance variables
private cas_cleared : set of IF_Types`CasualtyDetails;

operations
-- Constructor --
-----
public Gold_Police : PoliceId * PoliceRank * Role * IF_Types`Location
                                         ==> Gold_Police

Gold_Police(pid, ra, ro, l) ==
(
  id := pid;
  rank := ra;
  role := ro;
  location := l;
  cas_cleared := {}
);

-- info_to_clear interface operations --
-----
public clearCasualtyDetails : IF_Types`CasualtyDetails ==> ()
clearCasualtyDetails(cd) ==
(
  if forall c in set cas_cleared & c.loc <> cd.loc
  then cas_cleared := cas_cleared union
    {mk_IF_Types`CasualtyDetails(cd.number, true,
                                true, cd.type, cd.loc)}
)
pre role = <Gold> and cd.number >= 0 and cd.verified and
not cd.cleared
post card cas_cleared >= card cas_cleared` and
forall c in set cas_cleared &
  c.loc = cd.loc => c.verified and c.cleared;

-- press_conf interface operations --
-----
public pressRelease : () ==> IF_Types`ClearedDetails
pressRelease () ==
(
  return clearForPress(cas_cleared);
)

```

```

)
pre role = <Gold>
post RESULT.cleared and RESULT.verified and
    RESULT.number <= totalCleared(cas_cleared);

-- Private operations --
-----
-- policy for determining what data to release to press
private clearForPress : set of IF_Types\CasualtyDetails ==>
    IF_Types\ClearedDetails
clearForPress(cds) ==
is not yet specified;

-- Auxilliary functions --
-----
functions
-- function to count the number of casualty details cleared by Gold
private totalCleared : set of IF_Types\CasualtyDetails -> nat
totalCleared(cds) ==
cases cds:
  {} -> 0,
  others -> let cd in set cds in
    cd.number + totalCleared(cds\{cd})
end;
end Gold_Police

```

A.3.2 Gold Ambulance

```

class Gold_Ambulance is subclass of Ambulance, info_to_gold

instance variables
private cas_verified : set of IF_Types\CasualtyDetails;

operations
-- Constructor --
-----
public Gold_Ambulance : AmbulanceId * AmbulanceRank * Role *
    IF_Types\Location ==> Gold_Ambulance
Gold_Ambulance(aid, ra, ro, l) ==
(
  id := aid;
  rank := ra;
  role := ro;
  location := l;
  cas_verified := {}
);

-- info_to_gold interface operations --
-----
public receiveCasualtyDetails : IF_Types\CasualtyDetails ==> ()
receiveCasualtyDetails(cd) ==
(

```

```

    cas_verified := cas_verified union {cd}
  )
  pre role = <Gold> and cd.number >= 0 and cd.verified and
    not cd.cleared
  post cd in set cas_verified and cd.verified and not cd.cleared;
end Gold_Ambulance

```

A.3.3 Silver Ambulance

```

class Silver_Ambulance is subclass of Ambulance, info_to_silver,
                                     info_to_verify

instance variables
  -- collection of verified casualty details
  private cas_verified : set of IF_Types\CasualtyDetails := {};
  private cas_estimates : set of IF_Types\CasualtyDetails := {};

operations

  -- Constructor --
  -----
  public Silver_Ambulance : AmbulanceId * AmbulanceRank * Role *
                           IF_Types\Location ==> Silver_Ambulance
  Silver_Ambulance(aid, ra, ro, l) ==
  (
    id := aid;
    rank := ra;
    role := ro;
    location := l;
    cas_verified := {}
  );

  -- info_to_silver interface operations --
  -----
  public receiveCollectedCasualtyDetails : IF_Types\CasualtyDetails
                                             ==> ()
  receiveCollectedCasualtyDetails(cd) ==
  (
    cas_estimates := cas_estimates union {cd};
    verifyCasualtyDetails(cd)
  )
  pre not (cd.verified) and not (cd.cleared) and cd.number >= 0;

  -- info_to_verify interface operations --
  -----
  public verifyCasualtyDetails : IF_Types\CasualtyDetails ==> ()
  verifyCasualtyDetails( cd) ==
  if forall c in set cas_verified & c.loc <> cd.loc
  then cas_verified := cas_verified union
    {mk_IF_Types\CasualtyDetails(cd.number, true,
                                cd.cleared, cd.type, cd.loc)}
  pre role = <Silver> and not cd.verified and cd.number >= 0 and
    not cd.cleared
  post exists c in set cas_verified &
    c.loc = cd.loc and c.verified and not (c.cleared) and
    forall cas in set (cas_verified\{c}) &
    forall cas1 in set (cas_verified\{c}) & cas = cas1;

```

```
end Silver_Ambulance
```

A.3.4 Bronze_Police

```
class Bronze_Police is subclass of Police, order_to_collect_info

instance variables
  -- casualty details collected by officer
  private cas_estimates : set of IF_Types\CasualtyDetails;

operations
  -- Constructor --
  -----
  public Bronze_Police : PoliceId * PoliceRank * Role *
                        IF_Types\Location ==> Bronze_Police
  Bronze_Police(aid, ra, ro, l) ==
  (
    id := aid;
    rank := ra;
    role := ro;
    location := l;
    cas_estimates := {};
  );

  -- Collect interface operations --
  -----
  public collectCasualtyFigures : IF_Types\Location ==> ()
  collectCasualtyFigures(loc) ==
    cas_estimates := cas_estimates union {getCasualtyEstimates(loc)}
  post exists c in set cas_estimates & c.loc = loc and
    not(c.verified) and not(c.cleared) and c.number >= 0;

  -- Operation to collect casualty estimates
  private getCasualtyEstimates : IF_Types\Location ==>
                                IF_Types\CasualtyDetails
  getCasualtyEstimates(loc) ==
  (
    let cas_no = collect(loc) in
    return mk_IF_Types\CasualtyDetails(cas_no, false, false,
                                       mk_token("", loc));
  );

  --operation to collect casualty numbers
  private collect : IF_Types\Location ==> nat
  collect(loc) ==
    is not yet specified;

end Bronze_Police
```

A.3.5 Bronze_Fire

```
class Bronze_Fire is subclass of Fire, order_to_collect_info
```

```

instance variables
-- casualty details collected by officer
private cas_estimates : set of IF_Types\CasualtyDetails;

operations
-- Constructor --
-----
public Bronze_Fire : FireId * FireRank * Role * IF_Types\Location
                                         ==> Bronze_Fire

Bronze_Fire(aid, ra, ro, l) ==
(
  id := aid;
  rank := ra;
  role := ro;
  location := l;
  cas_estimates := {};
);

-- Collect interface operations --
-----
public collectCasualtyFigures : IF_Types\Location ==> ()
collectCasualtyFigures(loc) ==
  cas_estimates := cas_estimates union {getCasualtyEstimates(loc)}
post exists c in set cas_estimates & c.loc = loc and
  not (c.verified) and not (c.cleared) and c.number >= 0;

-- Operation to collect casualty estimates
private getCasualtyEstimates : IF_Types\Location ==>
                                         IF_Types\CasualtyDetails

getCasualtyEstimates(loc) ==
(
  let cas_no = collect(loc) in
  return mk_IF_Types\CasualtyDetails(cas_no, false, false,
                                     mk_token(""), loc);
);

--operation to collect casualty numbers
private collect : IF_Types\Location ==> nat
collect(loc) ==
  is not yet specified;

end Bronze_Fire

```

A.3.6 Bronze_Ambulance

```

class Bronze_Ambulance is subclass of Ambulance, order_to_collect_info

instance variables
-- casualty details collected by officer
private cas_estimates : set of IF_Types\CasualtyDetails;

operations
-- Constructor --
-----
public Bronze_Ambulance : AmbulanceId * AmbulanceRank * Role *
                                         IF_Types\Location ==> Bronze_Ambulance

Bronze_Ambulance(aid, ra, ro, l) ==

```

```

(
  id := aid;
  rank := ra;
  role := ro;
  location := l;
  cas_estimates := {};
);

-- Collect interface operations --
-----

public collectCasualtyFigures : IF_Types`Location ==> ()
collectCasualtyFigures(loc) ==
  cas_estimates := cas_estimates union {getCasualtyEstimates(loc)}
post exists c in set cas_estimates & c.loc = loc and
  not(c.verified) and not(c.cleared) and c.number >= 0;

-- Operation to collect casualty estimates
private getCasualtyEstimates : IF_Types`Location ==>
  IF_Types`CasualtyDetails
getCasualtyEstimates(loc) ==
(
  let cas_no = collect(loc) in
  return mk_IF_Types`CasualtyDetails(cas_no, false, false,
    mk_token(""), loc);
);

--operation to collect casualty numbers
private collect : IF_Types`Location ==> nat
collect(loc) ==
  is not yet specified;

end Bronze_Ambulance

```