Grant Agreement: 287829
Comprehensive Modelling for Advanced Systems of Systems

Model-based Engineering for Systems of Systems: the COMPASS Manifesto

Technical Note Number: Manifesto
Version: 2.0
Date: November 2014
Public Document

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

John Fitzgerald, UNEW
Simon Foster, UY
Claire Ingram UNEW
Peter Gorm Larsen, IHA
Jim Woodcock, UY

Editors:

John Fitzgerald, UNEW

Reviewers:

COMPASS Theme 2,3 leads
Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>22-03-2013</td>
<td>John Fitzgerald</td>
<td>Draft initiated</td>
</tr>
<tr>
<td>0.2</td>
<td>22-08-2013</td>
<td>Claire Ingram</td>
<td>Additional related work</td>
</tr>
<tr>
<td>0.3</td>
<td>14-10-2013</td>
<td>Jim Woodcock</td>
<td>Additional material on dynamic semantics</td>
</tr>
<tr>
<td>0.4</td>
<td>16-10-2013</td>
<td>John Fitzgerald</td>
<td>Restructuring, included tech foci material</td>
</tr>
<tr>
<td>0.5</td>
<td>18-10-2013</td>
<td>Jim Woodcock</td>
<td>Updated semantic material</td>
</tr>
<tr>
<td>0.6</td>
<td>20-10-2013</td>
<td>Peter Gorm Larsen</td>
<td>General Updates</td>
</tr>
<tr>
<td>0.7</td>
<td>28-10-2013</td>
<td>John Fitzgerald</td>
<td>Summarising case studies</td>
</tr>
<tr>
<td>0.8</td>
<td>29-10-2013</td>
<td>John Fitzgerald</td>
<td>Final updates</td>
</tr>
<tr>
<td>1.0</td>
<td>29-10-2013</td>
<td>John Fitzgerald</td>
<td>V1.0</td>
</tr>
<tr>
<td>1.1</td>
<td>10-08-2014</td>
<td>Peter Gorm Larsen</td>
<td>Added collaboration and tool support updates</td>
</tr>
<tr>
<td>1.2</td>
<td>14-11-2014</td>
<td>John Fitzgerald</td>
<td>Updates to reflect project late stages</td>
</tr>
<tr>
<td>1.3</td>
<td>14-11-2014</td>
<td>John Fitzgerald</td>
<td>Revised details of related projects</td>
</tr>
<tr>
<td>1.4</td>
<td>14-11-2014</td>
<td>John Fitzgerald</td>
<td>Updated conclusions on CPSs</td>
</tr>
<tr>
<td>2.0</td>
<td>17-11-2014</td>
<td>John Fitzgerald</td>
<td>V2.0</td>
</tr>
</tbody>
</table>
Abstract

A “system of systems” (SoS) is a group of largely pre-existing systems that, individually, may be independently owned and managed, but collectively are relied upon to deliver a service that emerges from their collaboration. Many SoSs have significant ICT elements; examples range from infrastructure such as integrated health services to cyber-physical systems formed from arrays of mobile devices. In spite of growing experience in creating and maintaining SoSs, the discipline of Systems of Systems Engineering lacks well-founded methods and tools that permit model-based development. Significant challenges are posed by the need to verify emergent properties in spite of the independence and heterogeneity of the constituent systems. The project “Comprehensive Modelling for Advanced Systems of Systems” (COMPASS) aims to develop rigorous approaches that will, for the first time, permit the machine-assisted formal verification of emergent behaviour based on SoS models that describe the behaviour of constituent systems in a contractual manner, allowing their continued independence to be taken into account. The approach is to develop a unified formal framework for the dynamic semantics of constituent system models, and build tools that exploit this to allow the construction of architectural models in notations supported by current industry practice.

This manifesto identifies the technical foci of COMPASS: the verification of emergent behaviours, support for collaboration and negotiation through contractual modelling styles, and the heterogeneity of dynamic semantics. It places our research in the context of related work in model-driven methods and looks forward to future directions in, for example, the engineering of cyber-physical systems.
## Contents

### I Challenges of Model-based SoS Engineering

1 Introduction 7

### 2 Dimensions of Systems of Systems

3 Applications

#### 3.1 Emergency Response

#### 3.2 Audio/Video Ecosystems

#### 3.3 Road Traffic Management

#### 3.4 Smart Grid

### II The COMPASS Approach

4 COMPASS in a Nutshell 25

5 Technical Foci

#### 5.1 Verifying Emergence

#### 5.2 Collaboration and Negotiation through Contracts

#### 5.3 Dealing with Semantic Heterogeneity

6 Tools for Model-based SoS

#### 6.1 Architectural Modelling in SysML with Artisan Studio

#### 6.2 Detailed SoS Modelling in CML

#### 6.3 Test Automation with RT-Tester

#### 6.4 Extending Tool Support with Plugins

#### 6.5 Tool-support for Collaboration

#### 6.6 Agreeing on Collaboration

### III Context: Related and Future Work

7 Related Work

#### 7.1 Model-driven Methods

#### 7.2 Related Projects

8 Future Directions 62
Part I

Challenges of Model-based SoS Engineering
1 Introduction

A System of Systems (SoS) is a group of independently owned and managed systems that together offer an emergent service [45, 40]. SoSs are typically distributed, and may evolve as a result of changes in their operating environment or in the goals of the owners of the autonomous constituent systems [14]. Examples include infrastructure such as smart grids and transport networks, as well as applications in healthcare, emergency response, and defence. As networking capabilities improve, the range of SoSs increases and reliance comes to be placed on the emergent behaviour that they offer. This manifesto argues that the successful engineering of SoSs demands methods and tools that allow this reliance to be justified.

Systems of Systems Engineering (SoSE) is a branch of systems engineering that addresses the challenges inherent in developing and maintaining SoSs. It emphasises cross-disciplinary thinking, the role of socio-technical aspects such as governance, business aspects such as procurement and the management of multiple diverse stakeholders. There is a lively SoSE community[1] and the European Commission has contributed to the development of the field [30], supporting two projects on model-based methods (COMPASS[2] and DANSE[3]), an agenda-setting project on transatlantic cooperation (T-AREA-SoS[4]), a roadmapping project (ROAD2SOS[5]), and four further projects launched in 2013: CPSOS (Towards a European Roadmap on Research and Innovation in Engineering and Management of Cyber-physical Systems of Systems[6]) AMADEOS (Architecture for Multi-criticality Agile Dependable Evolutionary Open System-of-Systems[7]) LOCAL4GLOBAL (Systems-of-systems that act locally for optimizing globally)[8] DYMASOS (Dynamic Management of Physically Coupled Systems of Systems)[9]. Nevertheless, although lively, SoSE is a young discipline, and general principles and patterns remain to be discovered.

The SoS engineer faces significant challenges that stem from the independence

---

1See, for example, the IEEE Conferences on Systems Engineering (http://www.ieeesyscon.org) and SoSE (http://www.sose2013.org); INCOSE (http://www.incose.org), which has an active SoS working group; the IEEE Systems Engineering Journal and the Journal of System of Systems Engineering.

2http://www.compass-research.eu
3http://www.danse-ip.eu
4http://www.tareasos.eu
5http://www.road2sos-project.eu
6http://www.cpsos.eu
7http://amadeos-project.eu
8http://local4global-fp7.eu/
9http://www.dymasos.eu
of constituent systems, the need for evidence to justify the reliance placed on emergent properties of the SoS, and the diversity of stakeholders in SoS development [26]. Model-based methods are seen as a way of addressing these aspects [20]. However, they do not provide the ability to incorporate heterogeneous constituent systems at the level of dynamic semantics, nor do they provide tool-supported methods to provide assurance of emergent properties.

Debates over whether a given system is or is not SoS, or whether a given engineering technique is special to SoSs do little to help practitioners to construct, maintain, evolve and understand real SoSs. Instead, our priority in the COMPASS consortium is to provide technical basis for model-based SoS Engineering that allows contract-based development, permits verification of emergence, and deals with the semantic heterogeneity that comes from this application domain. Our approach is informed by formal methods that exploit rigorous mathematical semantics for computer-based languages [63], allowing models to be constructed at levels of abstraction appropriate to each stage in the development process, and subjected to machine-assisted analysis, or to enable the exploration of design alternatives by permitting trade-off.

**COMPASS Technical Foci**

The COMPASS project aims to establish a basis for model-based SoS engineering that can support formal verification of emergent properties of a proposed SoS architecture in which constituent systems are described contractually (with the freedom to operate independently). We start from a view of system as an entity that interacts with other entities that form its environment. The common frontier between a system and its environment is the *system boundary* [9]. A system has an *interface* that defines the interactions of interest between the system and its environment. The *behaviour* of a system is the collection of traces or sequences of events that may occur at the system boundary. An SoS is a system composed of units (termed *constituents*) which are themselves systems. In an SoS, the environment of each constituent is composed at least in part of other constituents, and interfaces define the interactions between constituents. We focus on three problems that must be addressed in order to realise fully the benefits of model-based SoSE technology:

**Verification of emergent behaviour:** In our work we regard a behaviour as emergent if it can be observed at the SoS boundary. In contrast with usage in some disciplines, we do not assume that emergent behaviour is unanticipated or undesirable. For example, consider an SoS in which several banks produce smart card electronic cash systems that allow amounts of money
to be paid between cards. Developers might wish to verify the emergent property that the total amount of money in the SoS does not increase. The verification of such an emergent property requires the composition of the properties of the constituents.

**Collaboration and negotiation through contracts:** Constituent systems are provided and operated by stakeholders independent of the SoS as a whole. These stakeholders have the freedom to choose what information to reveal to and what to withhold from their peers in the SoS. In particular, they may change the goals and operation of constituent systems during the life of the SoS. As a consequence, the SoS engineer may not expect to be able to model constituent system behaviour completely. Instead, the constituent system must enter into a *contract* with its fellows. Modelling techniques must therefore permit the specification of *ranges* of behaviours that each constituent can rely on and guarantee. We therefore require the ability to verify that one constituent delivers behaviour that conforms to the expectations of others (we term this verification of conformance).

**Semantic heterogeneity:** SoSs and their models have many facets, including functionality, mobility, physical, human and socio-economic aspects. We regard all of these as vitally important to a full discipline of SoS engineering, but our project is focused on the computational aspects. Engineering for the interactions between these facets requires a multidisciplinary approach [27], but the models themselves, being rooted in different disciplines, may not share a common semantic framework, making it impossible to perform analyses, such as trade-offs, that cross the boundaries. For example, we may have constituent systems that have components that are synchronous and others that are asynchronous; some may have discrete state or time and others continuous state or time. There may also be different levels of abstraction. For example, a socio-technical medical SoS may have patients undergoing courses of treatment that last for months, while their medicines must be taken daily and their adaptive pacemaker must be accurate to 100ms. The semantic heterogeneity of this range of models must be addressed if emergent properties are to be verified with a level of automated support.

This manifesto presents the COMPASS project’s approach to the challenges of SoSE outlined above. The present version is the second edition, but we will maintain it as a living document after the project, as our methods and tools develop. A first version of the core material on the three technical foci was published as a keynote paper [35].

There are three main parts to the manifesto. In Part I, we motivate our work, introducing our basic concepts (Sect. 2) and giving the dimensions along which we
characterise SoSs, before presenting several of the project’s motivating examples in terms of these dimensions and briefly outlining the challenges that they pose to each of our three technical foci (Sect. 3). In Part II we present the COMPASS framework and tools in outline (Sect. 4) before discussing the approach taken to the three technical foci: the verification of emergent behaviour (Sect. 5.1), collaborative development through contracts (Sect. 5.2), and the apparatus required to deal with semantic heterogeneity (Sect. 5.3). Key to our approach is the principle that techniques should be realised in tools, and these are discussed in Sect. 6. Part III we first place COMPASS in the context of important related work in model-driven methods (Sect. 7), and we conclude by identifying challenges for future research (Sect. 8).
2 Dimensions of Systems of Systems

There is a substantial literature on the characteristics of SoSs. It is notable that authors tend not to draw a sharp distinction between SoSs and regular systems. Rather, SoSs are viewed as systems that have particular features that may indicate the use of specific engineering approaches. Perhaps the most widely cited characteristics are those first identified by Maier [45], and those of Boardman and Sauser [14]. Building on these, our review of the literature suggested a finer-grained set of characteristics that may affect the selection and use of formal model-based approaches [52]. We use these as a framework for characterising and comparing SoS engineering problems, and so term them “dimensions” as they each be present to greater or lesser degree in a given system.

**Autonomy:** the extent to which a constituent system’s behaviour is governed by its own rules rather than by others external to the constituent. Given the heterogeneity of an SoS, there is likely to be considerable variation in the autonomy exhibited by constituents. Modelling and analysis techniques need to permit the expression of a range of actions that an autonomous constituent may perform, but which may not be precisely predicted at the SoS level. This suggests that there is a need for looseness or underspecification of constituent system behaviour.

**Independence:** the capacity of constituent systems to operate when detached from the rest of the SoS. Independence implies that a given constituent system may offer a range of behaviours, some related to its role in an SoS, and others independent of it. The relationship between these classes of behaviour, and specifically the dependencies between them, might be hidden from the SoS engineer. Model-based techniques therefore need to be able to support information hiding.

**Distribution:** the extent to which constituent systems are dispersed so that some form of connectivity enables communication or information sharing. Modelling frameworks that support distribution require the ability to assign constituent system processes to computational infrastructure, linked by a communication medium. Descriptions of concurrency, communication, and particularly failures of communication media, are necessary.

**Evolution:** SoSs are long-lasting and subject to change, whether in the functionality delivered, the quality of that functionality, or in the structure and composition of constituent systems. Model-based approaches to SoS engineering require support for gaining assurance of the preservation of specified properties under evolution steps. We may characterise this as a need for ver-
ification of conformance of a constituent system’s interfaces to those of the other constituents with which it must interact. Evolution may be manifested as updates to constituent systems, requiring re-verification of conformance.

**Dynamic behaviour:** the capacity of an SoS to change its structure and composition, typically without planned intervention. This is in contrast to evolution, which refers to the capacity to support planned changes through intervention. Examples include on-the-fly swap-in and a pluggable architecture. To support dynamic behaviour, SoS models must have abstractions for the dynamic modification of architectures and interfaces, and the capacity to reason about such changing structures.

**Emergence of behaviour:** Emergent behaviours arise as a result of the synergistic collaboration of constituents, and are observed at the SoS boundary. Reliance is typically placed on the delivery of some emergent behaviours or on the absence of some undesirable emergent behaviours caused by, for example, feature interaction between constituents. From the perspective of model-based engineering, significant reliance on emergent behaviour brings a need to verify emergent properties by composition of the properties of constituents, or conversely the refinement of abstract specifications to models of the responsibilities assigned to constituents.

**Interdependence:** the mutual dependency that arises from the constituent systems having to rely on each other in order to fulfil the common goal of the SoS. If the objective of a constituent system depends on the SoS, then the constituent system itself may have to sacrifice some of its individual behaviour in order to meet the requirements of joining SoS. Modelling and analysis techniques should allow the explicit identification of interdependence, the tracing of mutual dependencies, and the ability to use these links to assess the impact of constituent system changes.

**Interoperability:** the ability of the SoS to incorporate a range of heterogeneous constituent systems. This involves the integration and adaptation of interfaces, protocols and standards to enable bridging between legacy and newly designed systems. The need for interoperability places several requirements on modelling and analysis methods, and reinforces the need to verify conformance of constituent system interfaces. Models of SoSs exhibiting a need for interoperability are likely to incorporate heterogeneous models of the constituents. Mechanisms for ensuring semantic consistency of diverse models, and rigorous analysis of those very distinct model types, are required in order to meet the needs for verification of both emergence and conformance.
3 Applications

SoS technology has many potential applications. Here we describe four motivating examples that are being used as case studies in COMPASS. We briefly describe each of them, characterising them in terms of the eight dimensions of SoS described Sect. 2 and indicating what each case study offers against our three technical foci.

3.1 Emergency Response

3.1.1 Summary

SoS engineering is becoming ever more important in the public sector where a “joined up” service needs to be delivered by systems that are owned and managed by different authorities. Our first study is based on emergency response – an application area that has become something of a benchmark in SoS research. The study is conducted by Insiel, a publicly-owned company in northern Italy that specialises in public sector systems and services. It concerns an Emergency Response System of Systems (ERSoS) that integrates systems such as hospital emergency room management software, ambulance and helicopter dispatch services, civil protection management systems, mountain rescue and fire brigade dispatch systems during specified emergencies. Although these systems are developed, managed and maintained independently, the ERSoS is expected to allow the systems to deliver a common workflow. The ERSoS recognises an event and, based on predefined rules, delegates control to relevant actors. Insiel is responsible for producing a constituent system – the call centre (“CUS”) – which manages incoming emergency calls by telephone, identifies available resources, and manages the emergency rescue units (ERUs) that are dispatched to the incident. The company has identified scenarios in which model-based analysis of the SoS would be beneficial. These include the addition of load monitoring to support alternative models of splitting running costs of the SoS; analysing the impact of a security protocol upgrade; and adaptation of one ERSoS to a new region with similar constituents, but different bases for integration. In all three cases, there is a need to be able to explore models of alternative architectures, verifying key properties of emergency service delivery.
3.1.2 Dimensions

We briefly consider the dimensions of this SoS. The constituent systems exhibit some autonomy. For example, an emergency service may change the range of services it offers (e.g., fire brigade may add a capability to rescue persons trapped in crashed cars), and a decision may be made to add/remove a constituent system or service by the CUS operator. Each emergency service can operate independently of the SoS. The CUS does not exhibit the same level of independence: without the participation of other constituent systems it cannot deliver any meaningful functionality. The constituents are geographically distributed both at the level of service headquarters and within the services themselves. The SoS may evolve in that the constituents may change and the relationships between constituents may also change, for example if they begin to share a significant resource (such as a helicopter). Goals and requirements of the SoS may also evolve under, for example, political pressure to change response times. In contrast, unanticipated dynamic reconfiguration is not regarded as desirable as it risks confusion on the ground, degraded response and gaps in records; acceptable reconfigurations are agreed in advance, for example to manage communication backups or service substitution.

The emergent behaviour of ERSoS is the delivery of a unified emergency service. This is not produced by the single constituent systems: for example, the police cannot deliver emergency medical support alongside policing services if they operate in isolation.

The interdependence of constituents is limited: the CUS, which interacts with most of the constituents, may need to adjust functionality if a decision made by a
service provider demands it, but other constituents are not able to sacrifice functionality in order to participate in the SoS. There are significant interoperability challenges: the constituents include elements of software (such as the CUS), hardware, third-party systems (such as a communications network, GPS maps and devices) and many diverse human participants. The constituents exchange information in widely varying formats, e.g. medical records or GPS coordinates.

3.1.3 Technical Foci

Verifying Emergence: There is a need to manage development and evolution risk by verifying significant emergent properties that can be set by technical or even politically-inspired needs. For example, a technical requirement to provide continuity of function in spite of the loss of communications to an ERU entails modelling the communications failure and backup mechanisms. Political requirements, such as a decision to require a change in response time, filters down to the ERSoS as a new requirement to dispatch an ERU within a specified interval from the arrival of a call, under certain assumptions about call rates. There is a need to evaluate whether such emergent properties can be met with a given architecture, and at what cost, in order to inform policy and contract negotiation.

Collaboration and Negotiation through Contracts: Contractual description of interfaces between constituents is central to the ERSoS study. For Insiel, “contracts” have a very real commercial as well as technical meaning. The need for a mobile and distributed technical solution implies relationships with highly independent suppliers of communications, as well as information systems in emergency services. Additionally, there are many factors - commercial, technical and political - that can lead to the need to renegotiate.

Semantic Heterogeneity: The ERSoS provides an opportunity to analyse a range of property types, including functional and temporal properties necessary to specify continuity of service in the presence of faults. There is potential to investigate performance-related properties such as call rates. Insiel have also identified challenges in the potential integration of cost models in considering alternative architectural solutions to evolutionary changes.
3.2 Audio/Video Ecosystems

3.2.1 Summary

Bang & Olufsen (B&O) develops and manufactures consumer electronics products in several areas:

- **Audio/Video**: High-end audio/video (AV) products with proprietary connectivity and SoS functionalities between AV and speaker products
- **Automotive**: Advanced car speaker systems, deployed in high-end car brands such as Aston Martin and Audi
- **Advanced speaker technology**: with distinctive features such as integrated amplifiers and acoustic lens technology
- **Play**: Youth brand products based on standard technologies and ecosystems like Airplay, DLNA, Android, iOS.

A B&O SoS can contain products from all four areas. The B&O “brand experience rule” (BER, written as $1 + 1 = 2 + n$) points out that the user with individual products enjoys the operational experience from each, plus a set of $n$ extended system experiences, emergent from the constituent products. Historically, B&O has used proprietary solutions to the interoperability challenges that flow from the BER. However, the business and technical demands to integrate multiple suppliers’ systems, and stream content from multiple providers, require a more open philosophy. Interoperability – and not only at a SCART or HDMI connector level, but transparent network interoperability – requires a shift from developing products to an SoS Engineering viewpoint.

In our case study, we examine a network of heterogeneous devices streaming digital content over a transport layer (shown in Figure 2). The SoS is composed of an audio product (Ax), a video product (Vx), a Play product, a gateway product (BLC), and a legacy audio product (ML). There is not a 1-to-1 relationship between product and constituent system: products can contain several constituent systems. Invisible constituents in this example include the audio streaming system, the network link control system and clock domain systems.

3.2.2 Dimensions

The individual constituents exhibit (potentially) wide variation in *autonomy*. They all operate at the behest of the user, but the fact that they may be legacy or non-B&O systems means that they may only offer a limited degree of controllability.
from the point of view of the SoS. The constituents exhibit operational *independence*; they provide stand-alone streaming or content browsing experiences, e.g. watching TV or selecting music to play. The constituents are typically *distributed* in different zones/rooms, the AV content can be local or remote, and the location of content source is often transparent to the user. Geographical distribution leads to *emergent* behaviours such as making sound follow the user around, driven by contracts between streaming and clock systems.

The constituents undergo *evolutionary* development. The stakeholders running non-B&O constituents like AirPlay, Digital Living Network Alliance (DLNA) and content providers, will have an evolution vision that is not necessarily compliant with that of B&O. In contrast to the emergency response case, there is *dynamic reconfiguration* behaviour in this SoS. Products join or leave the SoS during streaming or browsing operations, products can be turned off by users or by entering power-saving mode. While products have no *interdependence* for their stand-alone functionalities, a constituent may act as a source or a renderer during streaming or browsing operations, and thus constituents rely on each other in order to deliver the emergent behaviour that fulfils the SoS goal. The demand for *interoperability* is the motivating challenge for the AV SoS.

The level of managerial control is used as a basis for identifying design principles for SoS. However, it is important to recognise that the characterisation of the SoS as Virtual, Collaborative, Acknowledged or Directed may depend on the operating mode, and even the point of view. From a static point of view, if a centralised authority can control the actions of the constituent systems in subordination to the SoS goal, the B&O SoS is might be seen as directed. Applying a more dynamic
definition for the different states of the SoS suggests that the B&O SoS is collaborative, since the SoS has states where a designated manager is selected, and recognized objectives are agreed among the constituent systems.

### 3.2.3 Technical Foci

**Verifying Emergence:** The ultimate emergent behaviour here is the user’s experience of the delivery of audio/video content (as well as properties to do with the timely and reliable performance of control tasks). An important example from our recent work on this study in COMPASS is the need for an emergent leader election property for the underlying publish/subscribe architecture for streaming. Initial modelling and analytic work on this has already revealed a situation under which leader election behaviour is not emergent, and this was corrected in the commercial solution. Verifying failures of emergent behaviour prior to deployment is of utmost importance.

**Collaboration and Negotiation through Contracts:** As with Insiel, in the future B&O would like to be able to base their collaboration agreements on more well-founded, technically-oriented contracts.

**Semantic Heterogeneity:** For B&O the overall sound experience is of paramount importance and thus full synchronisation of speakers placed in adjacent rooms need to play music at exactly the same time. When all the speakers are manufactured by B&O that is easier to guarantee, but in an SoS setting one would like to be able to provide the same guarantee, even when some of the speakers are manufactured by other producers. A semantic combination of synchronous and asynchronous modelling is necessary to support this. Verification of temporal performance characteristics needs to be addressed alongside properties (such as deadlock freeness) that might be assessed by model-checking.

### 3.3 Road Traffic Management

#### 3.3.1 Summary

Road traffic management presents opportunities to examine widely distributed and evolving SoSs. In COMPASS, we work with West Consulting in the Netherlands (members of the COMPASS Interest Group) on developing a case study that exhibits these characteristics.
Traffic Control Centres (TCCs) are responsible for managing inter-urban road traffic, collecting traffic information and providing information to the public. A wide range of software systems is used for this purpose (usually distributed systems with components in the TCC itself and at the roadside). The organisation put in place to use these systems, and the division of control responsibility, suggest that the TCC can be regarded as an SoS. In many countries TCCs have been in development and use for decades, with the result that modern TCCs employ a wide variety of technology: some modern, some very outdated. The integration problems that occur because of this are the resulting costs. The UK Highways Agency\(^\text{10}\) and the Netherlands Rijkswaterstaat\(^\text{11}\) have started a project on the joint procurement of new TCCs aiming to reduce costs. Some legacy systems will be part of the new architecture, which is currently being defined.

The case study focuses on a subset of Traffic Management System (TMS) functionality related to ramp metering, as this affords an opportunity to examine competing goals for constituent systems, and particularly issues of fault tolerance and integration. Ramp metering systems (RMSs) control the entries and exits to high speed inter-urban roads. An RMS can decrease or increase the rate at which vehicles join the major road (Figure 3). It takes into account the current status of the major road (collecting data from induction loops) and whether there are vehicles waiting at the lights to enter the major road (collecting data from induction loops installed on the ramp). However, the joining rate must not be slowed so much that queues of traffic build up and spill onto surrounding access roads. An RMS may operate in one of several modes:

- Fixed-time mode, with predetermined duration of signals
- Adaptive mode, which responds to traffic conditions from a traffic monitoring device. This can be further divided into isolation mode in which the RMS responds to only local information about traffic status, or collaborative mode in which the TCC determines behaviour for all ramps in a stretch of road, using traffic monitoring data from a wide area.

Most RMSs in the Netherlands operate in isolated mode. However, in some situations a coordinated regional policy for admitting vehicles onto the major road with multiple access ramps will be more effective, and the RMSs go into collaborative mode.

\(^{10}\)http://www.highways.gov.uk/
\(^{11}\)http://rijkswaterstaat.nl/
3.3.2 Dimensions

The constituents of the TMS have autonomy in that they have individual goals and can operate and evolve independently to meet them. Distribution is inherent: elements of the SoS are mostly distributed roadside in a national network. Its long lifetime (many decades) means that evolution is also central to the TMS: the set of constituents changes over time as new technologies are developed and devices added to the TMS. The overall goals have also changed in the last 20-30 years, from a system that reacts to problems locally to one that proactively manages traffic in line with national or regional goals and policies.

Relations between constituent systems can be altered dynamically, e.g. where RMSs switch between operating modes in conjunction with other ramp meters on a sector. However, the TMS does not, for example, substitute one traffic monitoring system for another if the first becomes unavailable; this is because many sectors are covered by one type of sensor only. The functionality of the TMS (globally managing traffic) emerges only with the constituent systems working together. Collaborations between neighbouring TMSs is required for advanced planning, and a variety of systems are needed to enact management strategies. There is strong interdependency between the actuators (e.g. traffic lights, configurable speed limits) and traffic sensors. Without traffic-monitoring systems that can inform about current status, the actuators in the system cannot support global goals. Finally, interoperability is key: the constituent systems are heterogeneous and use a wide variety of hardware and data types, although standard data formats are generally used. Data are exchanged between constituent systems as well as between adjacent TCCs within the Rijkswaterstaat TMS, or with adjacent TMSs.
3.3.3 Technical Foci

**Verifying Emergence:** There is a need to verify both desired and undesired properties of emergent behaviours. In many cases unanticipated emergent behaviour can be very damaging. In preliminary modelling work, for example, we have demonstrated the possibility of previously unanticipated emergent properties following integration of a new version of a constituent RMS whereby the ramp metering system is not returned to the correct state after operating in a collaborative mode, for example, and therefore does not resume correct default behaviour.

**Collaboration and Negotiation through Contracts:** As with the verification of emergence, integration scenarios allow for exploration of the definition of the system by contracts. In particular they require analysis of the minimum information which must be made available via the contract for reliable collaboration to take place.

**Semantic Heterogeneity:** The challenge problem currently presents limited opportunities to explore the concept of semantic heterogeneity in depth. Although the constituent systems of the TMS produce widely varying data types, there are well-known data standards in common use which reduce the heterogeneity on the SoS. In the longer-term, a view of the SoS incorporating stochastic vehicle behaviour would be appropriate, but is beyond the scope of our current work.

3.4 Smart Grid

3.4.1 Summary

Smart Grid offers one of the most significant application areas for SoS engineering. Our example comes from collaboration with Grid Manager, another COMPASS Interest Group member company. Grid Manager provides energy management solutions through the deployment of smart energy meters, and analyses the data gained to generate energy-saving strategies. Different types of energy meters can be supported, including those developed by Grid Manager and third parties. Each meter type has specific phenomena it can measure (electricity, gas, water,
etc.) and different units (kWh, L, m$^3$, etc.). Some meters can also act as controllers, switching devices on/off according to defined rules (Figure 4). For example, a rule could be defined to the effect that all lights are turned off in a building when the main alarm is set. A rule may depend on several energy meters, in different locations, as well as other systems (e.g. an alarm system). The specific challenges Grid Manager identified were support for requirements management as well as modelling, analysis and simulation of large deployments of energy meters, and support to manage and guarantee the correctness of rules.

### 3.4.2 Dimensions

The proposed SoS is an energy management system. The constituent systems are the server, smart energy meters and other systems that may be used or controlled to manage energy usage (e.g. an alarm system). Although the energy management system has one overarching goal of achieving monetary savings, the different constituent systems are autonomous and must ensure safe operations independently. The server and the other constituent systems do not depend on each other for their correct operation, and so have independence. However, when they do cooperate improved operation is possible.

The constituent systems are physically distributed, communicating via the internet, and possibly via other forms of communication, such as SMS. The SoS is expected to evolve as the customer base grows and technology advances; if the SoS does not adapt to the customers’ changing needs it would become obsolete. On a shorter timescale, both the control rules and topology may change dynamically. The latter may happen when customers and constituent systems spontaneously join or leave due to network failures and repairs. The core functionality of the
energy management system is emergent: it depends on the constituent systems collaborating to achieve savings. The larger the SoS is, i.e. the more is known about energy consumption on a large scale, the better the predictions can be. In the energy management system interdependence only exists in the sense that the improved operation of the SoS depends on collaboration. However, it is required that all constituent systems also continue working independently. Interoperability only plays a role as far as the achievable service level is concerned. In cases where no computerised connection is possible, human operators can be notified, e.g. by SMS, in order to carry out certain actions.

3.4.3 Technical Foci

**Verifying Emergence:** There is a strong need to verify emergent properties here. The behaviour at the highest level is a desired change in power consumption profile and cost. There are more detailed behaviours relating to the performance of the SoS as a whole in: respecting time bounds for switching devices on or off; responsiveness to changes in sensed data; and overall safety constraints that can be violated by feature interaction between constituent systems.

**Collaboration and Negotiation through Contracts:** Grid Manager see a need for contractual modelling as a basis for a stronger technical process for interacting clients in the definition of the rules governing device management.

**Semantic Heterogeneity:** The Grid Manager case interacts with many different constituent systems with very different semantic entities. The heterogeneity and the interoperability required here needs to be supported by the rules engine for controlling the energy consumption of the different constituent systems. One of the most significant features of this example is the extent to which ICT can influence physical phenomena such as power loss or surge. This in turn suggests a need for integrated modelling approaches that combine models of the physics of power distribution with descriptions of controlling ICT elements: such models are typically very different in terms of the underlying semantics.
Part II

The COMPASS Approach
4 COMPASS in a Nutshell

COMPASS aims to develop rigorous, tool-supported foundations for SoS engineering that are strong enough to answer the variety of challenges that we encounter in applications like those discussed in Sect. [3]. The challenge we set for ourselves is that of developing a framework that addresses the three technical foci of verifying emergence, contractual description of constituent systems, and dealing with the deep semantic heterogeneity of models that are needed to support the exploration of SoS design alternatives.

As we indicated in Sect. [1] we approach our challenges from a viewpoint that emphasises the value of formalism. COMPASS brings established modelling techniques together in a new way and extends them specifically for the development of SoSs. This combined formalism is made accessible to a wide range of developers by providing different levels of description, including graphical architectural models using SysML. Architectural modelling patterns provide a basis for sharing experience in model-based SoS Engineering [38], while guidelines for requirements analysis and integration are intended to improve the deployment of modelling techniques throughout SoS life (e.g., [17]). Architectural modelling frameworks provide systematic approaches to the development of SoS model views for specific application domains [5]. SysML is linked to the new COMPASS Modelling Language (CML), which is founded on two very well-developed formalisms, Circus [62] and VDM [32], and extended with SoS-specific aspects such as features for the description of constituent systems in a contractual style. CML is intended to support a rich collection of static and dynamic analysis methods, including some forms, such as consistency checking, that deliver results which can be readily reflected back to the SysML architectural level. Other forms of analysis, including model-checking and symbolic proof, can be used to verify a wide range of emergent properties.

The vision for SoS analysis is ambitious, and requires a formal static and dynamic semantics for CML in order to enable the machine-assisted analysis that we seek. The semantics of CML are defined by means of the Unified Theory of Programming (UTP) [37] and the soundness of the analysis conducted at the CML level is justified in this framework. The overall approach is presented in Figure 5. Note that CML extensions could be developed for other modelling languages.

The COMPASS tools allow users to operate either at the SysML graphical level or at the textual CML level, or at a combination of these, since there is support for moving between these views. Sect. [6] describes the connection between the different tools. The tool set is split into a SysML base built on the Artisan Studio tool from Atego, and a CML base built on the Overture platform (itself built
on the Eclipse platform), which has already been constructed for VDM primarily by Aarhus University and Newcastle University. The CML platform is being extended with analysis plug-ins that permit the exploration of models by simulation, the generation and management of dynamic tests, and analysis of the dynamic semantics of models by means of model-checking and proof. The platform is open in that it does not depend on the use of Artisan Studio and links to other architectural modelling tools could in principle be developed.

Figure 5: COMPASS Formalism Vision.
5 Technical Foci

We have identified in Sect. 2 (and, in more detail, in [52]) eight dimensions of SoS. These dimensions pose particular challenges for Model-Based Software Engineering (MBSE) techniques. In our work on COMPASS we have identified three key problems introduced in Sect. 1 that we believe need to be addressed in order to fully realise the benefits of MBSE techniques for SoS engineering. These are:

1. Verification of emergent behaviour
2. Independence of constituents:
3. Semantic heterogeneity

Each of these are explained in more detail above in Section 1 and below we describe how we deal with each of these technical foci.

5.1 Verifying Emergence

Emergent behaviour can be observed in systems at all scales, including both complex systems and SoSs. A simple example can be found in Conway’s Game of Life, discussed in [57]. The game is played out as a two-dimensional cellular automaton with simple rules involving only individual cells and their eight immediate neighbours; an example rule might state that a live cell with fewer than two live neighbours dies of isolation. The rules are purely local, but non-local patterns emerge. The most famous such pattern is the glider: a group of five live cells that reconfigure according to the game’s rules, and in doing so appear to move as a unit in a particular direction. The glider, its direction, and its movement together constitute a global behaviour that emerges from the local rules: this observed behaviour has not been programmed anywhere. The glider is an example of weak emergence [11], where an emergent property can be reduced to its constituents; if we apply the local rules to a particular configuration of cells, the glider pattern emerges. This is in contrast to strong emergence, where an emergent property cannot be traced to any direct cause; an example would be consciousness emerging as a property of the brain. Like [57], we confine our attention to weak emergence.

The challenge that we address here is: how can emergence be verified in an SoS modelled in CML? This raises several questions which will be addressed in this section: firstly, how do we engineer an emergent property? And secondly, how do we analyse a system to check that an emergent property exists? Finally, we
also discuss in this section how COMPASS tackles the issue of verification in an evolving SoS.

### 5.1.1 Engineering Emergence

Sanders and Smith [57] demonstrate that it is possible to use formal methods to authenticate the trustworthiness of systems with emergent behaviour. The technique they use is to capture emergent behaviour in an overall system specification and then apply the incremental refinement method to justify design decisions embodied in an implementation.

Refinement plays an important role in COMPASS: one approach to verify the correctness of an SoS is to demonstrate that the CML refinement relation holds between two CML models. Using this approach, we can check whether an SoS has an emergent property or not. We start with a specification of the emergent behaviour and develop an implementation. Refinement guarantees that the property must hold in the refinement.

Our hypothesis is that certain emergent properties of a distributed system can be specified as global invariants. Here we have in mind global invariants that are purely specification artefacts: they do not exist in the implemented system. Our hypothesis applies equally to component-based systems and SoSs.

Our view is supported by Sanders and Smith [57], who employ the following motivational example. Suppose that we want to simulate the behaviour of a flock of birds implemented as autonomous agents. As in the Game of Life, the behaviour of the flock emerges from much simpler behaviour: each bird reacts autonomously to the behaviour of those birds in its immediate environment. Studying the flock as a unit, we can observe global behaviours to do with the position, volume and surface of the flock. These behaviours can be specified as abstract qualities. The correctness of the implementation (the model of the flock of individual birds) can be demonstrated by proving that the birds collectively refine the abstract specification [41].

To illustrate this approach in our framework, consider a simple model in CML. Mondex is an electronic purse hosted on a smart card and developed in the mid-1990s to the high-assurance standard ITSEC Level E6 by a consortium led by NatWest, a UK bank. Purses interact using a communications device and strong guarantees are needed that transactions are secure in spite of failures and attacks. These guarantees ensure that electronic cash cannot be counterfeited, although transactions are distributed. There is no centralised control: all security mea-
sures are locally implemented with no real-time external audit-logging or monitoring.

Once again, we notice a similarity with the Game of Life: autonomous smart cards play the role of cells and rules are purely local, with transactions involving only two cards out of the many millions in the overall SoS. The property that we want to emerge from the Mondex SoS is a global invariant, which in its simplest, idealised form can be expressed as constancy of value.¹³

Mondex was originally verified as follows. First the protocol for exchanging value between cards was modelled formally. Second an abstract specification was constructed with an Olympian view of the state of every card, with a global invariant expressing the constancy of value. The network of smart cards, with their purely local protocol, was then shown to be a formal refinement of the abstract specification. The global invariant has been completely refined away and no direct evidence of it is left in the network of smart cards. But we know, because we have formally verified it, that the global invariant still holds: the total value does not change in spite of transactions between cards.

Suppose that there are \( N \) cards, each initialised with \( V \) pounds sterling. The money supply, \( M \) is then \( N*V \). Cards are indexed from the set \( \text{Index} \), whose values are in the set \( \{1, \ldots, N\} \) and money comes from the set \( \text{Money} \), whose values are in the set \( \{0, \ldots, M\} \). We model communication between cards using channels, as follows:

\[
\text{channels} \\
\text{pay, transfer: Index \ast Index \ast Money} \\
\text{accept, reject: Index}
\]

The first channel, \( \text{pay} \), is connected to the user, who can instruct a card using the communication \( \text{pay} .i.j.n \): card \( i \) should pay \( j \) the sum of \( n \) pounds. The second channel, \( \text{transfer} \) connects card \( i \) to card \( j \) and tries to transfer the sum of \( n \) pounds. The third and fourth channels are used to signal to the user whether the transaction was accepted or rejected.

A contractual description of the interface to a card would be given as a CML process contains a state-based part and a reactive part. The card process is described in Figure 6. The state-based part contains the data and functionality alluded to in Sect. 5.2. The process is indexed by \( i \) and begins by declaring its encapsulated state: the natural number \( \text{value} \). It defines three operations on the state: 

¹³In reality, the electronic cash in the system can change via approved routes as cards are destroyed and money enters and leaves the system through banks.
process Card = val i: Index @
begin
  state value: nat
  operations
  Init: () ==> ()
  Init() == value := V
  Credit: nat ==> ()
  Credit(n) == value := value + n
  Debit: nat ==> ()
  Debit(n) == value := value - n
  actions
  Transfer =
    pay.i?j?n ->
    ( [n > value] & reject!i -> Skip
      []
      [n <= value] &
      transfer.i.j!n -> accept!i -> Debit(n) )
  Receive = transfer?j.i?n -> Credit(n)
  Cycle = ( Transfer [] Receive ); Cycle
@ Init(); Cycle
end

Figure 6: CML model of the Mondex card process.

Credit, and Debit\[14\]

The state value is initialised by Init to the value \( V \). The Credit and Debit operations increase and decrease value, respectively, by the sum \( n \). The protocol part of the contractual description is given by the definition of two reactive actions, Transfer and Receive, that use the operations to handle outgoing

---

\[14\]The purpose of this example is to illustrate an approach to verifying emergent behaviour, rather than showing all the features of CML. The operations are simple and are defined explicitly as single assignments. CML provides program-like constructs for defining operations explicitly. However, it also supports an implicit style in which operations are specified contractually by means of postconditions that express the properties of the state after the operation has executed. For the Credit operation, rather than giving the assignment, we might define a postcondition that \( value = value' + n \), where \( value' \) is the state variable before execution of the operation. Whether explicitly or implicitly defined, operations may also be constrained by preconditions that record the conditions assumed to hold before the operation is invoked.
and incoming payments, respectively. Transfer is triggered by the user communicating on the pay channel: pay.i?j?n. This is a communication to card i to make a payment; the value of the receiving card is bound to the variable j; and the value of the payment is bound to the variable n. The subsequent behaviour of the card is described by an external choice guarded so that, if the payment exceeds the funds available (n > value), it is rejected and the action terminates. If the payment can proceed (n <= value), then the transfer is made, and the action terminates.

The subsequent behaviour of the card is described by the external choice:

\[
\begin{align*}
  & (n > \text{value}) \land \text{reject}!i \rightarrow \text{Skip} \\
  & (n \leq \text{value}) \land \text{transfer}!.i.j!n \rightarrow \text{accept}!i \rightarrow \text{Debit}(n)
\end{align*}
\]

The branches of the choice are guarded so that they depend on the funds available. If the payment exceeds the funds available (n > value), then the payment is rejected and the action terminates. On the other hand, if the payment does not exceed the funds available (n <= value), then the transfer is made to the receiving cards, the payment is accepted, and the action terminates.

The cards are brought together as constituent systems in a network, and channels are connected as described above:

\[
\text{process Cards} = \\
\quad || i: \text{Index} @ \\
\quad \quad \{ \{ \text{pay}.i, \text{transfer}.i, \text{accept}.i, \text{reject}.i \} \} \\
\quad \quad \text{union} \\
\quad \quad \{ \text{transfer}.j.i.n \mid j: \text{Index}, n: \text{Money} \} \} \text{Card}(i)
\]

\[
\text{process Network} = \text{Cards} \setminus \{ || \text{transfer}! \}
\]

This is a model of a rather homogeneous SoS: although cards are independently owned and managed, they all follow the same rules. The real Mondex system is much more elaborate. For example, the transport medium is modelled; messages between cards can be corrupted, lost, or forged; and the power may go down at any point. In spite of these threats and faults, the property must emerge that all value is accounted for and so there can be no change in the value in the system.

We have a model of a simple SoS. How should we verify a specified emergent property of such a model? The specification for the Mondex system can also be given in CML, as in Figure 7. The specification describes the network of cards as a single, centralised process. It has an Olympian view of the state of every card, enabling it to assert global invariants. This is no way to implement the require-
process Spec =
  begin
    state
      valueseq: seq of nat
    inv
      len(valueseq) = N
      sum(valueseq) = M
    operations
      Init: () ==> ()
      Init() == valueseq := initseq(N)
    actions
      Pay = i,j: Index, n: Money @
        pay.i.j.n ->
          if n > valueseq(i) then
            reject.i -> Skip
          else
            ( valueseq := subtseq(valueseq,i,n);
              valueseq := addseq(valueseq,j,n);
              accept.i -> Skip
            )
      Cycle =
        ( |˘| i,j: Index, n: Money @ Pay(i,j,n) );
      Cycle
    @
  end

Figure 7: CML specification for the Mondex system.

ments for Mondex, which, roughly speaking, state that Mondex should behave like electronic cash: we don’t take our cash to a single common place in the world in order to pay for our groceries. But it is not meant to be an implementation, but rather the specification of an emergent property agreed upon by all the SoS stakeholders: constancy of value.

The global state is defined like this:

  state
    valueseq: seq of nat
  inv
    len(valueseq) = N
    sum(valueseq) = M

The state of each individual card is recorded as an element of the sequence valueseq.
accessed by the card’s index number. With all the state in one place, it is easy to specify our property: that all cards are accounted for and their total sum is the entire money supply. There is a single action, \textit{Pay}, that plays the part of the simple protocol in the implementation, transferring value atomically with assignments rather than \textit{transfer} operations.

We can now assert that the network of cards formally refines this specification. In the COMPASS tool environment, this assertion will be discharged either using the theorem prover or using the model checker.

Notice that the Mondex refinement can also be viewed as guaranteeing the absence of an emergent property: the system must not allow counterfeiting to emerge as a behaviour. Our technique of refinement allows us to confirm the presence or absence of an emergent property. We can describe the negation of an undesirable emergent behaviour in a centralised specification; a correct refinement guarantees that this undesirable behaviour is absent in the SoS.

5.1.2 Analysis of existing systems

Suppose that we have an existing system or system of systems; how do we discover and verify emergent properties? We describe some future work that addresses exactly this point. Simulation is often used as a technique to explore emergence; but of course, it is not a verification technique. In our future work in this area, we propose to use simulation of CML models to explore behaviours and to try to discover global system invariants.

Our inspiration is the Daikon tool for detection of likely program invariants \cite{29}. Daikon discovers invariants that hold for particular test-data sets, but which are not asserted in the particular program under consideration; in that sense, the invariant emerges from a dynamic analysis. Daikon has applications in documenting and understanding programs, generating further test cases, automating theorem proving and model checking, assisting component evolution and integration, repairing inconsistent data structures, and checking data validity.

As an example of the use of Daikon, consider a program that sums an array. The Daikon user would typically randomly generate 100 sample arrays, perhaps with exponential distributions of lengths and contents. The program would then be run on each of the array and its results monitored. The program is automatically instrumented to write data-trace files, which an offline invariant engine analyses, looking for potential properties amongst the data.

Daikon tackles two separate concerns: the choice of which invariants to infer; and
the inference of invariants. Daikon can infer only certain types of invariants that match choices from a library of patterns. The user usually specifies the invariant patterns of interest, leaving key parameters as symbolic constants. Daikon’s task then is to identify suitable values for these constants, discarding falsified putative invariants. It does this with a simple machine-learning technique. Interestingly, determining these constants can be quite inexpensive. The cost of inferring an invariant increases quadratically in the number of parameters, but is linear in the number of invariants to be checked, the test-set size, and the number of points in the program where invariants are being checked. As a final twist, Daikon supplies a confidence parameter indicating how likely it would be for a property to appear by chance in random input.

The major research challenge is to scale up the basic Daikon technique to large and multi-paradigm models. Currently, there is no Daikon-like system for inferring invariants in concurrent programs, object-oriented programs, pointer-rich programs, or for mobile or dynamically reconfigurable systems. But this challenge could be met by CML’s notions of contracts and invariants that cover these different paradigms, and by taking advantage of the natural parallelism of the invariant-inferencing technique.

5.2 Collaboration and Negotiation through Contracts

When a consortium of partners decides to work together in order to engineer a well-functioning SoS it is essential that they have proper support for collaboration and negotiation of what the different constituent systems shall deliver to the SoS. The basis of this can be established using formally defined contracts that can be analysed efficiently by tools. The stakeholders responsible for constituent systems in an SoS need to work collaboratively on reaching a common agreement that ensures that a given SoS design is feasible and has the desired functionality.

In practice today such agreements are: 1) not made at all; or 2) are made in the form of textual contracts by lawyers; or 3) are made directly at the coding level, e.g., with a service-oriented architecture. In the first case, developers have to work with whatever description they might have for the constituent systems. In the second case, the expense and inflexibility of legal contracts mean that the approach may only be expected to succeed for highly directed SoSs with limited evolution. In the third case, the SoS-level emergent behaviour may not be expressed at all, and the low abstraction level means that analysis capabilities may be limited.

In order to have a firm agreement on the collaboration there is a need to estab-
lish a type of contract that describes the connections and interactions between the constituent systems. Such a contract describes the interface a constituent offers to its environment in terms of the externally accessible services that can be called by other constituents, and the types for data that can be passed over the interface. The contract could be evolved to define more details and constraints on the services, data types and the sequence of events that the specific constituent system accepts.

In formalising such contracts it is worth considering for example the “design by contract” approach [47]. However, this needs to be augmented by the need to express contractual constraints of the proper protocol for interacting with constituent systems in a natural way. Here inspiration from temporal logic may be appropriate [23]. What is essential is that it must be possible to formally relate the contract for a constituent system interface to the internal (and potentially private) model of it. In a formal setting this is typically done using formal refinement [10].

Managing the information hiding aspects of collaborative development requires: a contractual modelling language that supplies abstraction from proprietary detail in addition to specific hiding constructs; and tool support to manage the collaboration during development. We expect to include features in CML for managing multiple models in the same SoS context, and specifically handling the explicit hiding of model elements. Given that the environment supports a rich group of analytic tools, it is not trivial to provide facilities such as verification support that retain the confidentiality of parts of models of constituents.

While studying a number of industrial cases we detected some recurring problems with certain aspects of specifying contracts in CML. To give future direction for discussion about patterns, evolution and emergent properties in CML, we propose an approach to modelling contractual obligations by implementation of generic contracts. A generic contract is similar to a process but is defined generically by parameterising it over the “concepts” which it requires to support, that can be implemented as higher-order variables.

Simple contracts with generic concepts can be seen as descriptions of Patterns. They can then be evolved by merging additional contracts in a way that preserves and extends all the properties of the previous contract. Contract state is not encapsulated, in the sense that the new contract can specify invariants over the state of the old one. Such patterns can provide the basis for building more concrete contracts, thus promoting reuse and enabling comparison between SoSs with similarities.

Many formal languages allow for their models to be described in modern Integrated Development Environments (IDEs), equal to does used in the software en-
5.3 Dealing with Semantic Heterogeneity

The architecture of an SoS and some of its Constituent Systems (CSs) may be modelled directly in CML, but it may not be suitable for all the CSs involved. There may be some domain-specific features of a CS that don’t fit easily into the CML paradigm: perhaps some CSs are synchronous and others asynchronous; some may use continuous variables or continuous time whilst others rely on discrete events and still others rely on the circuit-level description of a hardware device. It may also be that an SoS contains CSs that are already modelled using a wide variety of different, probably domain-specific, notations and different levels of granularity. So a serious attempt at modelling SoSs must address the question of semantic heterogeneity: how do we link diverse modelling paradigms?

The COMPASS approach to is to provide a common framework for capturing the semantics of the different modelling languages that are being used in an SoS, including CML itself. This common framework is Hoare & He’s Unifying Theories of Programming (UTP) [37]. The UTP Research Agenda has as its ultimate goal to cover all the interesting paradigms of computing, including both declarative and procedural, and hardware and software. Researchers have proposed many different theories of system development and practitioners have proposed many different pragmatic paradigms for industrial application; how do we understand the relationship between all of these? UTP proposes three principal ways:

1. **Computational Paradigms:** UTP groups programming languages according to a classification by computational model; for example, structured, object-oriented, concurrent, synchronous, real-time, and discrete or continuous values. The technique used to give semantics to each computational model is to identify common concepts and deal separately with additions and variations. UTP uses two fundamental and familiar scientific principles: simplicity of presentation and separation of concerns.

2. **Abstraction:** The description of each computational paradigm can also be categorised by different levels of abstraction to capture the development process leading from requirements through to code designed to run using the platform-specific technology of an implementation. Interfaces are specified using contracts to guarantee the correctness of moving a model from
one level to another. This mapping between levels is based on a formal notion of refinement that provides guarantees of correctness all the way from requirements to code.

3. **Presentation:** The third classification is by the method chosen to present a language definition. There are three well-known scientific methods.

   (a) **Denotational**, given by a function from syntax to a single mathematical meaning: its *denotation*. A specification is then just a set of denotations: the permitted behaviours of a system. Refinement is simply inclusion: every behaviour of the program must also be a behaviour permitted by the specification.

   (b) **Algebraic**, given by a collection of equations relating descriptions in the language. Interestingly, no direct meaning is given to the language at all.

   (c) **Operational**, given by a set of rules describing how descriptions in the language are executed on an idealised abstract mathematical machine.

As Hoare & He point out, a comprehensive account of constructing systems in any theory needs all three kinds of presentation. The UTP technique allows us to study differences and mutual embeddings, and to derive each from the others by mathematical definition, calculation, and proof. It presents a theoretical foundation for understanding software and systems engineering, and has already been exploited in areas such as hardware [55], hardware/software co-design [19], and component-based systems [65]. The paradigms used in the design of CML include theories for state-rich reactive systems with real-time, object-oriented, reference semantics, mobility of processes and their channel interconnections [61]. The relationship between CML and SysML is defined in UTP, with a comprehensive semantic definition of SysML and a bi-directional translation between CML and SysML. Other UTP theories are available to be included in CML if needed, including, but not limited to: hardware description semantics; synchronicity; and alternative computational paradigms, such as data flow, shared variables, and bulk-synchronous parallelism.

As well as using CML directly, the COMPASS approach with UTP presents an opportunity to construct domain-specific languages with heterogeneous paradigms and techniques. Having studied the variety of existing programming languages and identified the major components of programming languages and theories, we can select theories for new, perhaps special-purpose, languages. The analogy here is of a theory supermarket, where you shop for exactly those features you need while being confident that the theories plug-and-play together. Inevitably, new domain-specific languages will need some features not already available, but the
alphabetised relational calculus in UTP is a powerful semantic modelling tool and we expect that it will cope with these new features. We have no reason to suppose that it will not.

5.3.1 The COMPASS Modelling Language

Currently, CML contains several paradigms, the first two of which have already been seen in the Mondex example presented in Sect. [5.1]

1. **State-based description.** The theory of *designs* in UTP provides a non-deterministic programming language with pre- and postcondition specifications as contracts.

2. **Concurrency and communication.** The theory of reactive processes in UTP provides a way of constructing networks of processes that communicate by passing messages.

3. **Object orientation.** The theory of object orientation in UTP is built on the theory of designs and provides a way of structuring state-based descriptions through sub-typing, inheritance, and dynamic binding, with mechanisms for object creation, type testing, type casting, and state-component access.

4. **Pointers.** The theory of pointers in UTP provides a way of modelling heap storage and its manipulations, as found in implementations of object orientation, for which we have a reference semantics. Crucially, it supports modular reasoning about the heap.

5. **Time.** The theory of timed traces in UTP supports the observation of events in discrete time.

5.3.2 Galois Connections

The semantic models mentioned in Section [5.3.1] are each formalised as sets of relations. For example, state-based descriptions are represented as pre- and post-conditions, which are familiar from languages such as VDM and B. In UTP, an operation to decrement a variable $x$, which must invariantly be positive, would be written as $\text{ok} \land x > 0 \Rightarrow x' = x - 1$. The precondition requires that $x > 0$ and the postcondition ensures that the after-value of $x$, written as $x'$, is exactly one less than the before-value of $x$, written $x - 1$. This pair of predicates is modelled as a single predicate (a relation) with two observational variables: $(\text{ok} \land x > 0 \Rightarrow \text{ok'} \land x' = x - 1)$. This is read as “if the operation is started (the observation $\text{ok}$ is true) and $x > 0$, then the operation must terminate (the observation $\text{ok'}$ is true)
and when it does, \( x' = x - 1 \) must be true”. Designs are organised into a lattice of relations ordered by refinement. At the bottom of the lattice is the aborting operation and at the top is the infeasible operation that can never be started. All other designs are somewhere in between. The process of correctness by construction starts by specifying the requirements for an operation as a design, moving upwards through the lattice in a series of refinement steps, until an implementation is reached. As usual, the specification is chosen to make the formalisation of requirements as easy and clear as possible, whilst the implementation is chosen to be executable on the chosen technology platform. Choices between alternative, correct refinements in this process are usually determined by non-functional requirements.

Mappings exist between the different semantic lattices, and some of these are shown in Figure 8.

Figure 8: CML semantic models.

These mappings can be used to translate a model in one lattice into a correspond-
ing model in another lattice. For example, the lattice of designs is completely disjoint from the lattice of reactive processes, but the mapping $R$ maps every design into a corresponding reactive process. Intuitively, the mapping equips the design with the crucial properties of a reactive process: that it has a trace variable that records the history of interactions with its environment and that it can wait for such interactions. A vital healthiness condition is that this trace increases monotonically: this ensures that once an event has taken place, it cannot be retracted—even when the process aborts.

But there is another mapping that can undo the effect of $R$: it is called $H$, and it is the function that characterises what it is to be a design. $H$ puts requirements on the use of $ok$ and $ok'$, and it is the former that concerns us here. It states that, until the operation has started properly ($ok$ is true), no observation can be made of the operation’s behaviour. So, if the operation’s predecessor has aborted, nothing can be said about any of the operation’s variables, not even the trace observational variable. This destroys the requirement of $R$ that says that the trace increases monotonically.

This pair of mappings form a structure known as a Galois connection. Galois connections exist between all the semantic domains mentioned in the last section. One purpose of a Galois connection is to embed one theory within another, and this is what gives the compositional nature of UTP and CML. As an example, consider again the embedding of designs within the theory of reactive processes: every design can be turned into a reactive process. One of the most important features of designs is assertional reasoning, including Hoare logic and Dijkstra’s weakest precondition calculus. It has been reported that assertional methods are used more than any other formal method. Assertional reasoning can be incorporated into the theory of reactive processes by means of $R$. Consider the Hoare triple $p \{ Q \} r$, where $p$ is a precondition, $r$ is a postcondition, and $R$ is a reactive process. We can give this the following meaning: $R(p \vdash q) \subseteq Q$. This is a refinement assertion. The specification is $R(p \vdash q)$ here the precondition $p$ and the postcondition $r$ have been assembled into a design; this design is then translated into a reactive process by the mapping $R$. This reactive specification must then be implemented correctly by the reactive process $Q$. In this way, reasoning with preconditions and postconditions can be extended from state-based operations to cover all operators of the reactive language, including non-terminating processes, concurrency, and communication. This is the foundation of the contractual approach used in COMPASS: preconditions and postconditions (designs) can be embedded in each of the semantic domains and bring uniformity through a familiar reasoning technique. In summary, semantic heterogeneity is achieved through using UTP to include new semantic domains within the COMPASS framework. New domains are built as lattices of relations and equipped with Galois connections to compose and map
models.
6 Tools for Model-based SoS

The independence of constituent systems and the breadth of stakeholder groups involved mean that SoS development is necessarily collaborative. There may not be a single organisation in control of the development or maintenance process, and the organisations developing constituent systems may work remotely, with only limited contact. Tools for model-based SoSE therefore need to be Collaborative Development Environments (CDEs) that support the joint production of models that can still be systematically analysed, even though a development team might not want (or be able) to disclose every part of their model [15]. The tools therefore need to support private and public parts of the model and a mechanism for extracting the public parts of the model which still should be usable for other organisations. Tool support also need to support negotiation at a technical level where updates to interfaces between constituent systems can be proposed and accepted or rejected by the other parties.

In COMPASS, we aim to develop a CDE supporting the proper engineering of SoSs by independent stakeholders using a model-based approach called Symphonic [24, 50, 51]. Current frameworks for SoS modelling and analysis support simulation, e.g. [56, 31] or static network analysis, e.g. to support dependability assessment [36]. We believe that complementary analysis techniques are essential for being able to properly deal with the problems mentioned above. At the high level, a tool that supports an architectural notation like SysML is necessary to describe the abstract structure of the SoS and to clarify the requirements and capabilities of the constituent systems. At a detailed level, we require a tool that supports formal modelling of constituent systems in order to verify constituent system behaviour, emergent SoS behaviour, and the generation and verification of implementations. At this detailed level, we need to build confidence and insight into an implementation’s behaviour by providing test automation for the constituent systems and their interaction in the overall SoS.

We also note that our proposed tool is, itself, an SoS in miniature. The three specific software packages detailed in the sections below are each an independent system in their own right. The COMPASS platform links different tools packages addressing SysML modelling, formal CML model-based analysis, and test automation. Figure 9 shows the three main packages. In addition there are a number of additional external tools that has been incorporated. Communication between the packages may be by the use of flat files (all three packages will be able to read and write CML source files), or by direct communication to allow for synchronisation of their user interfaces. Architectural modelling in SysML is supported
6.1 Architectural Modelling in SysML with Artisan Studio

To meet the high level modelling need the COMPASS project uses Artisan Studio [8]. Studio already supports system and requirements modelling using SysML as well as software modelling using UML and code generation. COMPASS uses Studio’s well established extension mechanisms to extend traditional systems modelling as needed to model SoSs, and support the options to add any detail that may be necessary to unambiguously generate a CML model. Studio’s TDK (Transformation Development Kit) will be used to develop a new code generator that

---

Fig. 9: Overview of the COMPASS Toolset, Symphony

by Artisan Studio[^15] extended to allow the generation of CML from SysML, and synchronising changes between the two. Construction, maintenance and analysis of CML models is supported by the Symphony tools implemented on top of the Eclipse-based Overture Tool platform [44]. Test automation is supported by the RT-Tester tool which identifies test cases, traces them to requirements and generates concrete test data by means of an integrated constraint solver [54].

[^15]: Artisan Studio now implements Atego Modeler [http://www.atego.com/products/atego-modeler/](http://www.atego.com/products/atego-modeler/) For the purposes of this report, we refer to the older name.
will transform such an augmented SysML model into the textual format of a CML model, and also to roundtrip changes between the SysML and CML models.

6.2 Detailed SoS Modelling in CML

To provide tooling for the detailed level, the COMPASS project is developing a language called CML, using the deep experience of its members in the VDM state-based formal method and the CSP/Circus process-based formal methods. CML will combine the best elements of these formal methods and be implemented on the Overture Tool platform [44] (itself based on the Eclipse tool platform).

Figure 9 indicates the components planned for the COMPASS tool. CML source files of the detailed SoS model are parsed and the Abstract Syntax Tree (AST) is type-checked, ensuring a basic level of validity of the model. At this point the AST is available for use by plugins supporting a range of analytic techniques: the usual user interface elements; various static analysis plugins; and various simulation and dynamic analysis plugins.

The components in the Symphony tool suite vary in levels of maturity. CML is intended to provide abstract models of SoSs. Consequently, the size of a CML model may not directly reflect the scale of the underlying SoS. COMPASS tools’ scalability is restricted in the same way as that of other tools performing analysis of models written in languages as expressive as CML.

6.3 Test Automation with RT-Tester

For test automation tasks the RT-Tester tool is used as a starting point; it is being enhanced with respect to SoS-specific test and simulation techniques during the COMPASS project. Using formal concurrent real-time models of expected System Under Test behaviour as input, RT-Tester identifies test cases, traces them to requirements and generates concrete test data by means of an integrated constraint solver [54]. As a crucial capability for SoS testing, the current RT-Tester version already supports model-in-the-loop tests (test data is exercised against a simulation of the model), software tests and hardware-in-the-loop tests, the latter being important for checking the appropriateness of the HW/SW integration.
6.4 Extending Tool Support with Plugins

The software packages above will provide the core functionality required to support SoS development. However, large parts of the overall goals for tool support will be provided by plugins for the software packages. This allows us to implement the necessary functions in a modular manner and, where appropriate, make the functionality available for (re)use in other contexts.

6.5 Tool-support for Collaboration

The Symphony tool allows for connections to be established directly between tool instances to support the collaborative development of models in which distributed subsets of models are being connected to form the entire SoS model. As the constituent systems in an SoS most often are independently owned and developed by distributed development teams, a SoS focused tool must incorporate these characteristics. The tool must be able to work with other globally dispersed instances of itself. This enables a complete SoS model that is made up of distributed sub models describing the individual constituent systems.

As an SoS is established by a range of individual constituents systems interacting and working together on a common goal, the concept of collaboration groups is introduced to associate the development teams that have their systems involved in the SoS. The members of the collaboration group are considered as the stakeholders that can have an influence on the SoS design and development.

Each development team will have the model of their own constituent system encapsulated in their instance of the tool, and then through the tool decide which parts of the model they want to share with the rest of the SoS developers. As the tool is controlling the degree of visibility for the distributed sub model, system owners and developers will no longer have to fear that the internal implementations of their systems become widely visible.

Now, while the different instances of the tools may be able to communicate, they still need to agree on how the constituent systems are going to collaborate within the SoS. This is a process that requires human involvement.

6.6 Agreeing on Collaboration

The owners and development teams of the various constituent systems in an SoS need to work collaboratively on reaching a common agreement that ensures that
a given SoS design is feasible and has the desired functionality. Thus, in order to have a firm agreement on the system being developed there is a need for establishing a type of interface that describes the connections and interactions between the constituent systems. As a minimum such an interface describes the basic functionality that a specific constituent system offers to the rest of the world. As such the basic interface means the operations and the associated parameters that can be called by other constituents in the SoS. As the basic interface includes parameters, an agreement on the representation of different data types must also be described in the interface. The interface could be evolved to define more details and constraints on the operations, data types and the sequence of events that the specific constituent system accepts. The sequence of events could for instance describe some required login process, or the scenario of an initialisation operation that needs to be called prior to other operations. In the context of this paper these interfaces will be described in the CML modelling language and these descriptions will function as a sub model of the constituent systems and the connections between them. As such the combined sub models being shared between the different collaborators will form the entire SoS model.

The overall structure of the CDE approach is illustrated in Figure 10.

![Figure 10: Overview of the relations between Symphony instances](image)

The CDE plug-in contains four major features:

- Connectivity between CML models in separate Symphony (Eclipse) instances.
- Exchange and negotiation of interfaces between constituent systems for defining their collaborative relationship, as well as the properties of the SoS.
- Evolution of models such that the complete SoS model will be able to evolve over time, by allowing the interfaces and sub models elements shared be-
tween constituents to be renegotiated.

- Strong potential for extensions as it is built on an extensible open source platform with a powerful plug-in architecture.
Part III

Context: Related and Future Work
7 Related Work

7.1 Model-driven Methods

An SoS engineering problem may well involve models, designs and implementations expressed in a multiplicity of languages. Model-driven engineering, particularly metamodelling, addresses the relationships between models developed in design, within and across technical domains. While COMPASS aims to address heterogeneity within dynamic semantics, it deals only with models expressed in SysML and CML. The integration of COMPASS’ results into a multi-model environment thus builds on work in both areas. In this section, we discuss the interface between COMPASS and model-driven engineering in three significant areas: multiple technical spaces (Sect. 7.1.1), approaches to crosscutting concerns (Sect. 7.1.2) and model evolution (Sect. 7.1.3).

7.1.1 Dealing with Multiple Technical Spaces

The use of models and meta-modelling techniques have now become practically ubiquitous in software engineering. They form an instance of the more general technique of model-driven engineering [58], where complexity is tamed by being factored into different levels of abstraction, with a clear separation of concerns all the way from conceptual models down to the specific technology of target platforms. This chain of models is created linkwise, with each model being transformed into its successor, with the opportunity to add new models and new functionality at each stage. Technical spaces [43] have been proposed as a way of organising the concepts being used in these model chains and as a way of linking and distinguishing the concepts used in different technical approaches. A technical space is a body of knowledge with attendant modelling and transformation tools and techniques [53]. Examples of technical spaces include programming languages (e.g., Java and C#), database systems, frameworks for markup languages (e.g., XML, SGML), knowledge representation techniques (e.g., ontologies), and modeling frameworks and languages (e.g., UML, Model Driven Architecture (MDA)) [12].

The study of technical spaces is motivated [12] in a similar way to Hoare & He’s UTP [37]. First, although technologies may differ quite radically, they may share some common characteristic features. Studying these commonalities and differences in a general framework can reveal much about different technical spaces. Secondly, different technical spaces may offer complementary features, allowing
a problem to be solved in an easier or more elegant way by combining techniques to best advantage.

In the literature (e.g., [43, 12, 53]), technical spaces are organised across a layered architecture of model, meta-model, and meta-meta-model. For example, the technical space of the Java programming language has Java programs as its basic models, the grammar of Java programs as its meta-model, and Extended BNF as its meta-meta-model. The technical space for Java programs gives the context in which to understand these basic models. In a different technical space, such as the one for the C# programming language, the basic models and the meta-model are different, although the meta-meta-model is the same.

[43] introduces the important concept of bridging between different technical spaces to transfer artifacts between different contexts. This is accomplished by two kinds of projectors that transform between technical spaces. Suppose that we want to bridge between $J$, the technical space of the Java programming language, and $C$, the technical space of the C# programming language. Viewed from $J$, the projector from $J$ to $C$ is called an injector, and the projector from $C$ to $J$ is called an extractor. Apparently, it is often the case that an injector-extractor pair is needed to solve a problem [43].

The process of building bridges is an important and non-trivial topic [12]. In COMPASS, we address this problem by using UTP and Galois connections, as described in Sect. 5.3. Although the emphasis in UTP is on semantics and semantic heterogeneity, the analogy with technical spaces is clear.

Consider the semantic technical space of pre- and postcondition contracts, known as designs in UTP [37]. The models in this semantic technical space are individual designs: either the contracts for computations, or the nondeterministic programs that implement them. The meta-model is the theory of designs, including the signature that gives the syntax for designs and their nondeterministic programs, the alphabet of observations and program variables, and the healthiness conditions for designs. The meta-meta-model is the methodology of Unifying Theories of Programming, with its alphabetised relational calculus, which is used to define the theory of designs (the meta-model).

The relationship between different UTP theories, such as designs and reactive processes, is expressed as a Galois connection. This is the bridge between the two semantic technical spaces, with the pair of adjoint functions analogous to the pair of projectors between technical spaces. The notion of a bridge can be generalised to describe not only transformations, but embeddings and compositions.

The concept of a technical space is well developed, with a significant community of both researchers and practitioners. This provides a powerful technique for de-
scribing the static semantics of individual technologies and their inter-relationships. COMPASS aims to make a contribution to this community by extending the concept to a more extensive account of the semantics of these technologies and the translations between them. An interesting challenge would be to marry technical spaces and ontologies (as in [53]) with the UTP approach.

7.1.2 Dealing with Crosscutting Aspects

Aspect-Oriented Software Development (AOSD) focuses on the identification, specification, and representation of cross-cutting concerns and their modularisation and automated composition into a working implementation [42]. The most compelling advantage of AOSD over other software development techniques lies in its particular support for the key abstraction principle in software engineering: separation of concerns. The idea is to separate different aspects of a system’s requirements and then program them separately; an intelligent compiler then weaves the aspects together into a single program. New aspects can then be added one by one without needing dramatic modification of the program.

In considering the aspect-oriented paradigm in CML, it is natural to follow the process algebraic metaphor [4]. Aspects in this metaphor are described as processes and the semantics and combinators of process algebra are used to support the development and manipulation of aspects. Although the formalisation of the aspect paradigm is relatively recent, the idea has its origins in the modelling of the conjunction of requirements as the parallel composition of processes [64], which in turn was inspired by Jackson’s System Design method [39]. Consider the development method used in engineering Safety-Critical Java (SCJ) programs from Circus models [21]. Each requirement is captured individually and formalised as a Circus process. The specification is the conjunction of all the requirements and is implemented by the parallel composition of the corresponding Circus processes. This simple structure imposed by the decomposition of requirements is unlikely to be an efficient or appropriate architecture for implementing a real-time program, so the system description is normalised by eliminating the parallel composition operator using the algebraic laws of Circus. The resulting sequential process can then be decomposed using refinement into the chosen architecture.

The metaphor equates an aspect to a process capturing a single requirement. A join point is a specification of when, in the corresponding main program, the aspect’s code should be executed; this is equated to event synchronisation between different processes (aspects). This technique works if synchronisation can involve multiple partners, not merely two, and if the synchronisation is open, so that more processes can be added. This is the case in CSP and its relatives Cir-
and CML. Communication between aspects is naturally modelled by communication between processes. Finally, aspect weaving is the normalisation of parallel processes into a single sequential processes. Concurrent aspect programming then corresponds to the further refinement step to a particular distributed architecture.

Consider a small example taken from [49] that involves a protocol for sending a message in a network. The requirements state that the sender must connect to the network before sending a message; the sender must disconnect from the network after sending; and connecting and disconnecting must strictly alternate, starting with connecting. Here we have three aspects that can be specified separately:

\[
\begin{align*}
\text{CONNECT} &= \text{connect} \rightarrow \text{send} \rightarrow \text{CONNECT} \\
\text{DISCONNECT} &= \text{send} \rightarrow \text{disconnect} \rightarrow \text{DISCONNECT} \\
\text{SEQUENCE} &= \text{connect} \rightarrow \text{disconnect} \rightarrow \text{SEQUENCE}
\end{align*}
\]

The specification of the aspects that the program must satisfy is the parallel composition of these three processes:

\[
\text{PROTOCOL} =
\begin{align*}
( \text{CONNECT} & \mid \{\text{send}\} \mid \text{DISCONNECT} ) \\
& \mid \{\text{connect,disconnect}\} \mid \\
\text{SEQUENCE}
\end{align*}
\]

The three aspects can be woven into a single, sequential program by rewriting using the laws of CML producing the following:

\[
\text{PROTOCOL} = \text{connect} \rightarrow \text{send} \rightarrow \text{disconnect} \rightarrow \text{PROTOCOL}
\]

Further aspects can still be added. For example, it may be decided that an acknowledgement is required after sending, but before disconnecting. This can be specified in the obvious way and added in parallel with the \text{PROTOCOL} process. This can then be woven into a new, purely sequential process; there was no need to start from scratch.

7.1.3 Dealing with Evolution

One property that has been repeatedly associated with an SoS [2, 13, 25, 28, 45] (and identified as one of our SoS dimensions [52]) is a process of continuous evolution. During evolution, new properties emerge, and so our progress here is linked to support for emergence and contractual specification.
Evolution may induce new or altered emergent properties in the SoS. Such emergent properties could be detected by observation of the interaction between properties of the underlying contracts. Verification of an evolved system then involves tackling two challenges:

- Can we verify continued *conformance*? For example, does a new or evolved CS still conform to the obligations placed upon it by the environment of other CSs?
- Can we verify that *emergent behaviours* are still satisfied by a new or evolved CS?

The COMPASS approach to verification of emergence and of conformance is presented in Sect. 5.1, where we argued that continued conformance of a CS to its contract guarantees emergent SoS behaviour.

The evolution of an SoS can be viewed relative to the evolution of contracts between the constituent systems. Constituent systems must agree on an initial contract which can then be modified by evolution transformations as desired. The evolution in the contract must be mirrored by a documented transformation on each constituent system. Then it should follow that the contract transformation is refined by the constituent system transformation. We apply this process continuously as evolution is required, whilst retaining the SoSs development history. We can therefore look back to the previous design decisions which were made and maybe rollback to a previous contract, should our requirements in the future change. The COMPASS approach to contractual specification was presented in Sect. 5.2.

### 7.2 Related Projects

In this section we explore the outcomes of some other European projects investigating SoS as well as projects investigating complex systems, and outline how COMPASS builds on these outcomes.

#### 7.2.1 SPEEDS

SPEEDS (Speculative and Exploratory Design in Systems Engineering)\(^\text{16}\) was a European Project (2006-2010) running in conjunction with the ARTEMIS\(^\text{17}\) Joint

\(^{16}\)http://www.speeds.eu.com/

\(^{17}\)http://www.artemis-ju.eu/
Undertaking, which co-ordinates research into Embedded Computing Systems. SPEEDS aimed to ‘reduce product development time and cost of complex embedded systems in key European industrial sectors’ [60]. The project focussed on [1]:

- Semantic-based modelling to support design of complex, heterogeneous embedded systems and integration of tools;
- New formal techniques for exploring architectural decisions and ‘design by contract’; and
- A new design process, centred on a trustworthy development environment and definition of contracts, supported by tools.

Some key challenges addressed by SPEEDS were: cross-cutting requirements such as safety and reliability; challenges presented by concurrent design teams in a multi-disciplinary environment; and cross-organisational developments and ensuring the co-operation of manufacturers and suppliers. The project produced a design process and a chain of integrated tools to support it, and adopted Model-Based Software Engineering (MBSE) as a possible solution for managing complexity and reducing ambiguity [1] in cross-domain and multi-dimensional teams. The SPEEDS development process employs a component-based design process, introducing Heterogenous Rich Components (HRC) [1]. HRCs are described by formal contracts, thereby allowing a design-by-contract paradigm and verification in the early design stages. Contracts describe functional and non-functional component aspects, and can be organised using differing viewpoints to give cross-dimensional, cross-cutting views of a project. SPEEDS particularly focused on viewpoints that present functional/behavioural, real-time and safety aspects [1] of a system. A contract is defined by SPEEDS as an assumption-promise pair [59]:

The assumption specifies some constraints on the environment (expected property of the environment) in which the component operates, and the promise specifies what the component will guarantee provided the environment does not violate the given assumption [1].

The COMPASS tools and techniques build further on this concept of contractual specification, by extending it into a new field. Although there are some conceptual differences between an embedded system (the focus of SPEEDS) and an SoS (the focus of COMPASS), cross-organisational development is a key challenge in both areas. Contractual specification offers a potential solution, because:

- It permits formal analysis techniques to increase confidence in a system’s dependability, from an early design stage.
• It supports the information-hiding which is necessary when multiple commercial parties are involved.

• It facilitates modular, concurrent development teams in cross-organisational developments.

• It enables component replacement, if it is possible to define one contract as an acceptable alternative for another. SPEEDS uses the term *dominate*: ‘if a system component-contract is dominated by a library component-contract, then system component can be replaced by the library component without having to re-validate the entire system’ [59]. The SPEEDS development process is predicated on a component-based development paradigm, where components may need to be replaced, but this is necessary also for SoSE scenarios, where a CS may need to be replaced in the future if its development trajectory and long-term aims do not match the demands and needs of the SoS, or where dynamic reconfiguration may need to be employed to adapt to environmental changes.

COMPASS builds on the SPEEDS project’s approach of formal contractual descriptions by adopting a similar contractual specification to describe constituent systems (CSs) of an SoS, whilst augmenting the ‘design by contract’ paradigm. In COMPASS, a contract characterises a CS by formalising the expectations and obligations placed on its behaviour [18] (described in Sect. 5.2). In addition COMPASS explores contractual specification as a tool for facilitating the verification of emergent behaviour in an SoS (described in Sect. 5.1).

A key challenge addressed by both SPEEDS and COMPASS is the question of heterogeneity in representations of a system. SPEEDS tackles tool integration by developing a Heterogeneous Rich Components (HRC) model. This approach relies on a common interchange format that can be read and written by a variety of design tools. It is represented as the ‘SPEEDS bus’; each integrated tool communicates with the bus and is therefore required to implement the SPEEDS API. An advantage of this approach is outlined in a white paper by the SPEEDS project:

‘The introduction of the SPEEDS bus as one central means of interaction among development tools allows them to access process relevant data in a unified way... facilities such as central logging and event notification are realized... process metrics can be derived online and process advice is provided’ [1].

HRC modelling in the SPEEDS project demonstrated that it is possible (although not trivial) for design aspects from different teams to be integrated into a merged model; the HRC approach has been claimed to lead to a decrease of information
losses when merging or converting models between formats and tools [59]. However, there is a considerable initial effort and cost to achieve tool integration if all tools must implement a common API, and additionally each tool may only be integrated if the separate development teams accept this step. SPEEDS operates in an embedded systems domain, where many developments depend on the success of a supply chain; this provides incentive to move towards greater integration. In contrast, whilst an SoS may be capable of exhibiting similar dependencies between constituents, most SoSs do not resemble supply chains and will not exhibit similar influences between the constituent systems.

COMPASS builds on the experience of SPEEDS for the development of the COMPASS tool suite. We have selected an existing standard for data exchange between tools. This approach has advantages and disadvantages, but it allows COMPASS to by-pass the possible problems posed by point-to-point tool data exchange (also by-passed by SPEEDS), and avoids some of the problems posed by the selected SPEEDS solution by ensuring that the common exchange format is standardised.

7.2.2 CESAR

‘CESAR’ (Cost-efficient methods and processes for safety relevant embedded systems) [18] was a European project (2009-2012) running in conjunction with the ARTEMIS Joint Undertaking [19]. Many design and modelling tools or methods employed by the safety critical embedded systems industry address only one system aspect (e.g., safety), and are not integrated into a coherent process or interoperable with other tools. The CESAR project was initiated to redress this problem, aiming to:

‘bring significant and conclusive innovations in the two most improvable systems engineering disciplines:

- Requirements engineering in particular through formalization of multi viewpoint, multi criteria and multi level requirements
- Component based engineering applied to design space exploration comprising multi-view, multi-criteria and multi level architecture trade-offs’ [20]

18 http://www.cesarproject.eu/
19 http://www.artemis-ju.eu/
Increased formalization in the requirements stage enables early consistency checking, helps to elicit more details at an earlier stage, reduces ambiguity and misinterpretation and increases confidence in the fault tolerance of a system. The CESAR project advocated multi-views of requirements and cross-cutting aspects to ensure that greater consistency of non-functional behaviours (such as safety) can be achieved right across the supply chain. As with the SPEEDS project, CESAR concentrates on embedded systems; the project advocated component-based development paradigms to support independent, concurrent development teams.

CESAR’s major focus was on redressing ‘the lack of open and common interoperability technologies supported by the different tools that generate and provide access to the data’ in safety critical embedded system development [22]. The CESAR project developed a ‘Reference Technology Platform (RTP)’ to act as a standardisation initiative and to enable integration or interoperation of existing tools and technologies, specifically focussing on enabling data sharing:

‘The difference between sharing and exchanging of data is related to the underlying implementation approach. When implementing interoperability to exchange data, the work and information flows supported by the system are predefined, whereas the implementation of data sharing, allows a more generic access which is work and information flow independent’ [22]

Related work continues with the ARTEMIS project MBAT [21] which builds on the RTP developed by CESAR for the transportation domain to add a verification and validation reference technology platform (MBAT RTP). This will support high-quality and safe embedded systems produced in shorter timescales with reduced costs.

In CESAR’s domain of safety critical embedded systems, the lack of common, domain-agnostic interoperability technologies has become a problem that hinders future research. We anticipate that design work on a single SoS will incorporate data from a variety of tools and processes. COMPASS tools allow the SoS engineer to analyse collected models of constituent systems, which may be produced in a variety of domains, with differing levels of abstraction and different concepts or notations. The COMPASS approach to integrating tools conforms to the CESAR definition of ‘data sharing’ rather than ‘data exchange’ (which requires following predefined information flows). Like CESAR [22], COMPASS implements a common data format (described in Sect. 6) approach to ensure data sharing between the component tools under the COMPASS umbrella.

[21] https://www.mbat-artemis.eu/home/
On COMPASS the details and concepts which to be integrated between tools are encapsulated in CML, which in turn has a UTP-compliant semantics. This needs to be flexible enough to encompass a wide range of SoS domains with very different levels of abstraction, concepts and notations. In Sect. 5.3 we discuss how this will be addressed. The COMPASS approach is also intended to permit development of domain-specific or special-purpose languages, the interoperability of which we can be reasonably confident so long as they conform to the UTP-compliant semantics. This will allow the COMPASS technologies to express the needs of constituent systems from a variety of domains, as well as to capture cross-cutting constraints and system aspects. The COMPASS approach differs from that of CESAR, which necessarily considered each domain in turn and the tools already in use in that domain, the concepts it expresses and the level of abstraction it requires, in a field which is already populated with notations and tools. The CESAR project addresses tools drawn from a single domain; the expectations of the COMPASS project are a single SoS will straddle different domains.

7.2.3 CRYSTAL

CRYSTAL (CRitical sYSTem engineering AcceLeration) is a project running in conjunction with ARTEMIS, which began in May 2013. The project builds on previous ARTEMIS work in the field of safety critical embedded systems, with a particular focus on quality, cost-effectiveness and architecture platforms. The primary project goal is to improve reusability of both the technologies and processes involved in the development of safety-critical embedded systems, specifically in the domain of transportation. This will involve establishing workflows through improved system and safety analysis and tools, and development of new technological ‘bricks’ that will facilitate reuse. CRYSTAL employs as an interoperability standard the Reference Technology Platform (RTP) developed by the CESAR project (Sect. 7.2.2).

The CRYSTAL project is not yet developed enough for COMPASS to draw conclusions or lessons from early deliverables. There are some similarities between the concurrent embedded system design teams that are the focus of the ARTEMIS project and an SoS engineering team (discussed in Sect. 7.2.1), although it is likely that not all elements of the workflows optimised for embedded system teams, or the tools specifically designed for embedded system analysis in the transportation

---

23http://www.artemis-ju.eu/
24http://www.artemis-ia.eu/project/index/view?project=46
industry, will be transferable for the domains and SoSE fields on which COM-
PASS concentrates.

7.2.4 DANSE

DANSE (Designing for Adaptability and evolution in Systems of systems En-
engineering) is an EU 7th Framework Programme project running from 2011 to
2014. DANSE addresses unsolved challenges in the field of SoS engineering, par-
ticularly the management of unexpected emergence which could result in loss of
service or compromising of safety goals, as well as enabling an SoS to maintain
stability whilst achieving fast-changing global objectives and supporting stake-
holder optimisation decisions. The project’s goals are to develop new approaches
for SoS engineering, validated with real-life test cases. These will include:

- an SoS lifecycle methodology
- contractual modelling for SoS interoperations
- SoS architectural approaches to support non-disruptive integration of new
  or altered components
- tools to support activities for SoS analysis, simulation and optimisation

The DANSE approach to SoS modelling concentrates on providing different views
of the SoS, representing key elements such as local and global goals, the CSs and
their capabilities, and the relations between them. SysML or UML can be used
as a notation to represent the key concepts. An SoS View is provided to express
the properties which have been associated with SoSs by Maier, such as ‘ge-
ographical distribution, different roles depending on the operational scenario, the
ownership relation and support of changes on different timescales. A System
View is also provided to represent a single CS, with the view tailored to the needs
identified in the SoS view. CSs may play different roles within an SoS and both
provide and require specific capabilities.

Contracts and goals play a key role in the DANSE modelling approach:

‘Contracts are... defining assumptions of the CS to the SoS and the
physical environment and providing a promised behavior if the as-
sumptions are fulfilled... there are system level contracts as men-
tioned above and SoS level contracts which argue over the role of an

---

25 http://www.danse-ip.eu/home/

26 http://www.danse-ip.eu/home/index.php/about/main-project-objectives

individual CS in the SoS and the roles and relations of other systems in the SoS... The SoS goals should represent the combination of the local goals of the CSs and conflicts of local goals should be solved by the structure of the SoS if they are not changeable. It is assumed that the information on CSs throughout the SoS may be incomplete, so local views of the CSs may be derived from belief or observation. The approach allows misunderstandings or conflicting goals to be modelled explicitly.

The DANSE approach to contractual specification, the Goal and Contract Specification Language (GCSL), is an extension of a Contract Specification Language developed under the auspices of the SPEEDS project. GCSL offers a textual, pattern based specification notation, intended to complement requirements in natural language with formalisation; it uses contracts to describe architectural aspects of the overall SoS and goals or ‘optimization targets’. As with the SPEEDS project, the design-by-contract approach supports CS re-use and integration into new or evolved SoSs. Contracts can be checked for compatibility at an early stage, whilst goals allow the SoSE engineer to record target functions, or the local goals of the the CSs.

Like COMPASS, DANSE advocates formal modelling as a solution to some of the challenges of SoSE and contracts are a means for modelling interfaces between CSs and/or the SoS. COMPASS concentrates on verification of emergence; contractual specification; and coping with semantic heretogeneity. DANSE, on the other hand, concentrates on: integration; use of temporal logic to express changes over time; optimisation; and interactions between (possibly conflicting) goals and capabilities. In both projects, the contractual approach allows the modeller to cope with problems posed by independent CSs, such as the need to implement some information hiding or to accept observations of a CS’s externally visible behaviour in place of reliable information about its internal logic. For COMPASS, contract specification is a building block that allows further developments in the areas of: patterns; fault analysis; verification of new or newly evolved CSs; and verifying that unexpected interactions between proposed constituents will not produce unwanted behaviour. For DANSE it forms the basis of further work on: resolving goal conflicts; performing integrations with minimal disruption; optimisation; and checking for compatibility for new or replacement CSs.

7.2.5 The KeY Tool

The KeY tool project began in 1998 and is now a joint project of three institutions. The KeY tool supports an approach for the deductive reasoning of object-oriented software, providing a platform that allows conventional and formal software analysis and development methods to co-exist and collaborate as seamlessly as possible. The KeY tool itself is designed as a plug-in to two separate, widely-employed software development tools (Borland Together and Eclipse). Given a software development project consisting of specifications and an implementation, in either of these two tools, the KeY tool plugin can generate proof obligations from selected subsets of the specification, which can be verified using the KeY prover. For specifications KeY supports the OMG standard Object Constraint Language (OCL) and the The Java Modelling Language, translating from these notations into logic or rendering OCL expressions into fragments of natural languages. The verification target language for KeY is the Java Card, selected ‘because of its importance for security- and safety-critical applications’ (the target applications for the project).

Findings from the project may be useful to COMPASS, which seeks to integrate architectural and graphical modelling approaches (conventionally implemented in SysML) with formal modelling techniques (implemented in CML). However, the KeY tool is currently aimed directly at security- and safety-critical domains and the language features and tools that it supports are naturally aimed at this target area. In contrast, the COMPASS tools must present a high-level view for an SoS designer, or the owner of a CS which participates in and depends on the SoS behaviour. SysML, the existing notation selected by COMPASS, is already widely accepted by both systems engineers and SoS engineers, and the language features that are supported by CML are necessarily wide-ranging to cover the needs of a designer working with independent CSs that may not be able to share data or to rely on the behaviour or continued presence of the remaining CSs.

http://www.key-project.org/
8 Future Directions

Several aspects of SoS engineering provide promising areas of future research. Cyber-physical systems (CPSs) combine facets of embedded systems and SoS, integrating continuous-time or continuous-state descriptions of physical phenomena, with the discrete models of cyber elements [16]. The integration of semantics for co-analysis and co-simulation of models of cyber-physical systems brings with it opportunities for rapid exploration of the design space, making informed trade-offs across the cyber/physical boundary [34].

SoS and CPS engineering are essentially collaborative. From our experience in COMPASS, it is apparent that support for collaboration entails support for abstraction at constituent system interfaces, and a contractual approach has value here. In CPS engineering, the value of semantic foundations for exploring cyber and physical phenomena, and their consequences on emergent behaviour, come to the fore. A successful approach that permits the verification of emergent behaviour requires semantic support for multiple modelling paradigms; COMPASS UTP-based semantic framework is a first step. We would argue that the combination of a contractual approach to modelling and a sound approach to semantic heterogeneity has the potential for significant impact in engineering CPSs, although many challenges to fully realising this remain [33].

Our work has been motivated by dependability in the technical sense, but designing systems that command the less tangible property of trust from citizens demands more radical approaches. Their incorporation in a model-based framework would entail the integration of radically different models, including mobility and the potentially stochastic description of human behaviour. Our work on SoS Engineering methods has not so far addressed socio-technical and human factors, and in particular the need for human-centered design off SoSs and CPSs remains an open and intriguing area for potential research.

We have identified three fundamental issues that we believe must be addressed if model-based methods are to achieve their potential in the engineering of systems of systems: contractual specification, verification of emergence, and semantic heterogeneity. We have described the foundations of an approach, implemented in the COMPASS project, that seeks to address these by combining structured modelling in SysML with formal methods. Contractual description is enabled by a formal modelling language that combines the description of data and functionality with communication and concurrency, object-orientation and real time features. A notion of refinement underpins the verification of emergent behaviour.

Semantic heterogeneity is addressed by using UTP for the systematic integration of semantic domains.

It is important to stress the provision of guidelines and patterns for managing requirements, architectural modelling and integration in SoS. We expect that the provision of a semantically sound framework for SoS models expressed using SysML can lead to patterns that are amenable to machine-assisted verification. The link from SysML to CML has been outlined [48]. This has allowed us to describe, for example a profile for fault modelling in SysML models [6], leading to verification of fault tolerance on the CML models derived from them [7]. The focus in COMPASS is on linking the tools that automate parts of this process to provide a semantically sound chain. Within the project, case studies are being undertaken in networks of home audio/video devices and content streaming network, emergency response, traffic management and smart grid.

Acknowledgements

This work has been supported by the Commission of the European Communities under the project “Comprehensive Modelling for Advanced Systems of Systems” (COMPASS, Grant Agreement 287829). We are grateful to the EC, to Werner Steinhögl, and the project reviewers for discussions leading to this manifesto.

References


