A Systematic Approach to Model-based Engineering of Cyber-Physical Systems of Systems

A thesis by

Martin Mansfield

Submitted in partial fulfilment of the requirements

for the degree of

Doctor of Philosophy

School of Computing

Newcastle University

Newcastle upon Tyne, UK

June 2019

For Ashley

Acknowledgements

First and foremost, I would like to thank my supervisor, John Fitzgerald. I am eternally grateful for the privilege of his insightful advice, careful guidance, inexhaustible patience and unwavering encouragement, without which this thesis would not have been possible.

My gratitude owes also to my colleagues in the Cyber-Physical Lab at Newcastle University, not only for their invaluable advice over the last few years, but for creating an enjoyable working environment in which to study. Particular thanks are extended to Richard Payne, for his support with all things SysML, to Carl Gamble, for all of his help with continuous-time modelling, and to Ken Pierce, for guiding me through the world of co-modelling. Thanks also to Alexei Iliasov, for his advice throughout the development of a rail case study.

Special thanks go to my family. I am enormously fortunate to have had the unbounded support of my parents; none of this would be possible without them. Above all, sincerest thanks go to my partner, Ashley; her love and belief in me has been crucial in the completion of this work. Without her relentless support, encouragement, tolerance—and at times, nagging—this work would still be in its infancy.

Abstract

This thesis describes and evaluates methods for the model-based engineering of Systems of Systems (SoSs) where constituents comprise both computational and physical elements typical of Cyber-Physical Systems (CPSs). Such Cyber-Physical Systems of Systems (CPSoSs) use sensors and actuators to link the digital and physical worlds, and are composed of operationally and managerially independent constituent systems that interact to deliver an emerging service on which reliance is placed.

The engineering of CPSoSs requires a combination of techniques associated with both CPS engineering and SoS engineering. Model-based SoS engineering techniques address organisation and integration of diverse systems through the use of disciplined architectural frameworks and contractual modelling approaches. Advances in model-based CPS engineering address the additional challenges of integrating semantically heterogeneous models of discrete and continuous phenomena. This thesis combines these approaches to develop a coherent framework for the model-based engineering of CPSoSs.

The proposed approach utilises architectural frameworks to aid in the development of rich abstract models of CPSoSs. This is accompanied by the specification of an automated transformation process to generate heterogeneous co-models based on the architectural description. Verification of the proposed engineering approach is undertaken by its application to a case study describing the control of trains over a section of rail network, in which the (cyber) behaviour of control infrastructure must be considered in conjunction with the (physical) dynamics of train movements. Using the proposed methods, the development of this CPSoS uses architectural descriptions to generate an executable model to enable the analysis of safety and efficiency implications of the implemented control logic.

The utility of the approach is evaluated by consideration of the impact of the proposed techniques on advancing the suitability and maturity of baseline technologies for the engineering of CPSoS. It is concluded that the proposed architectural framework provides effective guidance for the production of rich architectural descriptions of CPSoSs, and that the conversion between architectural and executable models is viable for implementation in a suitable open tools framework.

Contents

List of Figures

1 Introduction

This thesis is a contribution to advancing the state of the art in utilising models in the engineering of Systems of Cyber-Physical Systems (CPSoSs). The next generation of transportation networks, energy generation, storage, and distribution systems, manufacturing plants, and buildings must respond to global challenges by improving energy and resource efficiency, reducing emissions and waste, and providing improved services at lower costs. To do this, such systems must integrate a vast collection of *'smart'* devices, capable of sensing their environment, performing complex computation, and communicating with other devices. These infrastructures constitute CPSoSs, and their inherent complexity makes them uniquely challenging to design and maintain. In particular, the organisation of a collection of independently owned and managed components, each of which tightly couples the cyber and physical worlds, makes this new class of system especially difficult to engineer, and demands the use of models in their design. To aid the engineering of CPSoSs, an approach is proposed which utilises complementary modelling techniques and technologies in a systematic way to overcome the challenges particular to them.

This chapter introduces core concepts used throughout the thesis, outlines the scientific contributions made by the proposed research, and summarises the structure of the document. Section 1.1 gives an overview of how the use of models can be advantageous in addressing challenges facing systems engineers, and Section 1.2 includes a summary of the characteristics of systems of interest, as well as the particular challenges inherent to them. Section 1.3 identifies some of the limitations of existing approaches to engineering the systems of interest, and isolates the particular research challenges to be addressed by this work. Section 1.4 provides a summary of objectives necessary for overcoming the identified challenges and outlines the approach employed to achieve those objectives. The content contained within the remainder of the document is outlined in Section 1.5.

1.1 Model-Based Systems Engineering

A *system* describes “a collection of hardware, software, people, facilities, and procedures organized to accomplish some common objectives” Buede & Miller (2016). The *International Organization for Standardization* (ISO)[1] provide definitions of ‘system’ and related concepts in the standard *ISO 15288 – Systems and software engineering life cycle processes* (ISO/IEC/IEEE 15288 2015). They describe a system as “a combination of interacting elements organised to achieve one or more stated purposes”, and qualify that “a system may be considered as a product or as the services that it provides”.

There are a many complementary definitions of what constitutes a system, but major themes are consistent throughout. Systems typically describe a collection of interconnected elements which are able to produce results unobtainable by any of the elements acting alone, where each element might include people, hardware, software, policies and documents, collectively forming the necessary components to produce systems-level results, including qualities, properties, characteristics, functions, behaviour, and performance (ISO/IEC/IEEE 15288 2015, ANSI/EIA 632 2003, Walden et al. 2015, Hirshorn et al. 2017). Beyond the capabilities of each element acting independently, the value added by the system as a whole is created by the relationships between its elements (Rechtin 2000).

1.1.1 Systems Engineering

Systems Engineering is a discipline with a responsibility for creating and executing an interdisciplinary process to ensure that stakeholders’ needs are satisfied in a high quality, trustworthy, cost efficient and schedule compliant manner throughout a system’s entire life cycle (Bahill & Gissing 1998). Fagan (1978) traces concepts of systems engineering to Bell Telephone Laboratories in the 1940s, and describes major applications of systems engineering during World War II. The first use of the phrase “systems engineering” was in a memo in the summer of 1948, and systems engineering was identified as a unique function in the organisational structure of Bell Laboratories in 1951.

A significant authority on systems engineering is the *International Council on Systems Engineering* (INCOSE)[2], who describe systems engineering as “an interdisciplinary approach and means to enable the realisation of successful systems”, where a system is defined as “a combination of interacting elements organized to achieve one more stated purposes” (Walden et al. 2015). The *INCOSE Systems Engineering Handbook* from which these definitions are taken is based directly on ISO 15288. Together

[1]See `https://www.iso.org/`
[2]See `https://www.incose.org/`

with the Systems Engineering Research Center (SERC)[3], and the Institute of Electrical and Electronics Engineers Computer Society[4], INCOSE oversee the *Systems Engineering Body of Knowledge* (SEBoK) project[5]. SEBoK began in 2009 as part of the larger *Body of Knowledge to Advance Systems Engineering* (BKCASE) project[6], with an aim to provide a comprehensive set of resources for systems engineers. Rather than proposing additional, unique definitions, SEBoK collates existing systems engineering terminology and concepts in an effort to highlight best practice.

ISO 15288 outlines 14 technical processes and supporting activities which enable systems engineers to coordinate interactions between engineering specialists, other engineering disciplines, and system stakeholders and operators. The technical processes are intended to guide engineering activities throughout the life-cycle of a system, and their use helps to avoid project failure and leads to the development of requirements and system solutions which address desired capabilities in terms of performance, environment, external interfaces, and design constraints (Walden et al. 2015).

The systems engineering processes support the identification of system requirements and the transformation of requirements into an effective product, beginning with the development of *needs* (capabilities desired by stakeholders) and *requirements* (formal, structured statements that can be verified and validated) (Ryan 2013). A *business analysis process* together with a *stakeholder needs and requirements definition process* describe activities to establish business requirements from the strategic vision and goals of an organisation, and stakeholder requirements from operations stakeholders. Requirements engineers can then transform business and stakeholder requirements into system requirements under the guidance of activities described in a *systems requirements definition process*. A supporting *architecture definition process* outlines activities for the selection of a system architecture from a series of viable alternatives based on the established requirements.

A complementary series of systems engineering processes are concerned with the consistent reproduction of products which satisfy the requirements of a system. A *design definition process* outlines activities for supporting the definition of key system elements in sufficient detail as to enable an implementation which is consistent with the chosen architecture. The use of mathematical analysis, modelling, and simulation to support other technical processes is included in a *system analysis process*. System elements can be realised to satisfy system requirements, architecture, and design, and then combined to produce the designed system under the guidance of activities described in an *implementation process* and an *integration process*. A supporting *verification process* describes activities to demonstrate that the

[3]See `http://www.sercuarc.org/`
[4]See `https://www.computer.org/`
[5]See `http://www.sebokwiki.org/`
[6]See `http://www.bkcase.org/`

resulting system satisfies the specified requirements.

A final set of systems engineering processes support the utilisation of products for the sustained provision of services until the retirement of a system. A *transition process* details activities for the deployment of a system into operations, and an *operation process* and *maintenance process* outline activities for sustaining a system throughout its operational life. A supporting *validation process* describes activities to demonstrate that a system will achieve its intended use in its intended operational environment. Finally, a *disposal process* describes activities for the deactivation, disassembly, and removal of a system from operations after its retirement from service.

1.1.2 Model-Based Engineering

Model-Based Engineering (MBE) describes an approach to engineering where models form an integral part of the requirements, analysis, design, implementation, and verification of a capability, system, and/or product. A *model* is an abstract version of a concept, phenomenon, relationship, structure, or system, and provides an abstract description of the reality of such entities (van Amerongen 2010). A model might be a graphical, mathematical or physical representation, and is abstract in that it only contains details relevant to the particular purpose for which it was constructed, eliminating unnecessary components. Models are typically used to facilitate understanding, to aid decision making, and to explain, control, or predict events.

Model-based approaches to engineering differ from document-based approaches in the way that information is captured, analysed, shared, and managed. In a document-based approach, information is typically distributed between a series of diverse artifacts, making it difficult to maintain, synchronise, and asses in terms of correctness, completeness, and consistency. In contrast, a model-based approach captures information in a model or set of models which are primary artifacts of the engineering process. In doing so, engineering teams are able to better communicate design ideas, more easily understand the impact of design changes, and analyse properties of a system design before it is built (Walden et al. 2015).

Model-based Systems Engineering (MBSE) describes the aspects of MBE specifically associated with systems engineering. Such aspects include system architecture, requirement traceability, behavioural analysis, performance analysis, simulation, and testing. INCOSE describes MBSE as “the formalized application of modelling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases" (INCOSE 2007).

By employing an effective MBSE approach, Holt & Perry (2008) suggest that the cost, time, and resources required to develop a system can be reduced. They propose that by supporting the provision of a single resource for accessing system information, such an approach can result in consistency across whole system architectures, traceability between system artifacts, measurement and control of system complexity, and the automated generation of system documents.

MBSE can lead to significant improvements in the quality of system requirements, architecture, and design, and lower the risk and cost of system development (Walden et al. 2015). A range of model-based techniques which might be employed for undertaking systems engineering processes and activities are outlined in Chapter 2.

1.2 Cyber-Physical Systems of Systems

Cyber-Physical Systems of Systems (CPSoSs) are large and complex systems, in which an array of physical elements are monitored and controlled by a distributed and networked series of computational elements. Within a CPSoS, physical systems are often spatially distributed and have significantly complex dynamics. Control of constituent systems is distributed between independently owned and managed authorities, and as such the CPSoS is subject to continuous evolution and its overall behaviour emerges from the interactions between constituents (Thompson et al. 2015).

CPSoSs exhibit characteristics typically associated with *Cyber-Physical Systems* (CPSs). These include the integration of many distributed physical subsystems which tightly interact with, and are controlled by, a network of distributed computing elements and human users. Furthermore, they also exhibit characteristics common to *Systems of Systems* (SoSs). These include the integration of a number of subsystems with varying levels of autonomy, continuous evolvution throughout their life-cycle, frequent and dynamic reconfiguration, and emergent behaviours.

CPSoSs fall within the intersection of CPSs and SoSs, and exhibit characteristics typical of both domains. In order to further establish the characteristics and challenges associated with CPSoSs, the remainder of this section separately considers the characteristics and challenges of CPSs and SoSs in detail. Challenges particular to CPSoSs are described in Section 1.3, where the specific challenges which this work aims to address are also identified.

1.2.1 Cyber-Physical Systems

In order to understand CPSs, it is important to understand the field from which they emerged. Since the earliest days of electronic computing, electro-mechanical devices have been augmented by the addition of an embedded computer. In contrast to a general purpose computer, an embedded computer system is contained within an electronic device to perform a dedicated set of tasks, often unnoticeable to the device user (Vahid & Givargis 2001). Computers replacing mechanical or human controllers were introduced as early as the 1940s, when computers were proposed for the control of chemical processes (Stout & Williams 1995).

Whilst embedded computing had been established, hardware was initially far too large to be practical to embed in more everyday items. Although the introduction of microprocessors in the early 1970s vastly increased the potential for embedded systems, their reliance on additional hardware to implement a working system prohibited economical computerisation of household appliances. By integrating the arithmetic and logic capabilities of a microprocessor along with the necessary ancillary circuitry on a single chip, the cost of embedding computers was drastically reduced. Augarten (1983) describes how this was first achieved in 1974 with the release of the first microcontroller: the TMS 1000, created by Texas Instruments.

Since their introduction, microcontrollers have become widely used in almost all aspects of modern life. Today, embedded systems have become commonplace in the home, motor vehicles, the workplace, and even the outdoors. A microcontroller can be customised for a particular function and added to almost any modern appliance (Toulson & Wilmshurst 2012). From washing machines to burglar alarms, and from engine management systems to vehicle braking mechanisms, embedded computing systems can be found in abundance in everyday modern life. With advancing electronics technologies enabling both smaller and more powerful embedded computers, the integrated computational ability of physical systems continues to grow.

As embedded computers become more sophisticated, the software running on them becomes increasingly integral to the system in which it is included. With cyber and physical elements becoming ever more tightly coupled, a new class of system emerged: CPSs. CPSs tightly couple the virtual and physical worlds and comprise software, computing hardware, and equipment to facilitate interaction with the physical environment. Within a CPS, an embedded computing and communication core monitors the physical environment through sensors, and performs complex computation based on sensed data. Computational elements coordinate, control, and integrate physical processes using actuators (Lee 2010, Rajkumar et al. 2010), interacting via feedback loops (Lee 2008).

Cengarle et al. (2013) suggest that the first use of the term "Cyber-Physical System" was in 2005, in the United States of America. Academics in Berkley coined the term in response to the evolution of a new kind of system, where consideration of both cyber and physical elements is essential to describing the overall performance of the system. Early definitions described CPSs as "physical, biological, and engineered systems whose operations are integrated, monitored, and/or controlled by a computational core . . . The computational core is an embedded system, usually demands real-time response, and is most often distributed" (Gill 2008).

As interest in CPSs developed internationally, a number of descriptions of CPSs and their characteristics emerged. Lee & Seshia (2011) distinguish CPSs from other disciplines by the requirement to understand the joint dynamics of computers, software, networks, and physical processes in order to design them. They describe that "as an intellectual challenge, CPS is about the intersection, not the union, of physical and cyber", and comment that "it is not sufficient to separately understand the physical components and computational components".

CPSs are more than simply networked embedded systems; they are computationally complex, intelligent systems, with the capacity to collaborate with other systems, adapt to their environment, and evolve over time. The emergence of CPSs concedes a paradigm shift from traditional embedded systems, where computational ability is not just added to some physical system, but the functionality of the system as a whole relies on the tight integration of cyber and physical.

An *Integrated Research Agenda* by the *German National Academy of Science and Engineering* provides a detailed definition of CPS (Geisberger & Broy 2012). Their definition describes a CPS as a system with embedded software, which:

- uses sensors and actuators to record and affect physical processes
- records and evaluates data, and actively or reactively interacts with the physical and digital world
- utilises networks to communicate with other CPSs
- utilises globally available services and data
- offers a number of human-machine interfaces.

More recently, the *Cyber-Physical European Roadmap and Strategy* (CyPhERS) project[7] led to a simplified and more general definition of CPS. Their definition specifies that "A CPS consists of computation,

[7]See `http://cyphers.eu/`

communication and control components tightly combined with physical processes of different nature, e.g., mechanical, electrical, and chemical" (Cengarle et al. 2013).

In order to effectively utilise embedded systems, it is often necessary for a number of engineers from a wide variety of fields to cooperate in their design (Verhoef et al. 2014). Specialists in complementary disciplines must work together throughout the design of embedded systems, from identification of an appropriate hardware platform, to implementing software to perform the necessary processing. For basic functions this can be trivial, but where computational tasks are more demanding, the heterogeneous nature of the systems becomes much more significant. For CPS engineering, this orchestration of cyber and physical is crucial (Lee 2008).

There has been significant research investment in the development of methods and tools to assist in engineering CPSs, from both the National Science Foundation[8] in the United States (Wolf 2007), and Horizon 2020[9] in Europe (Thompson 2013*a*). The *Strategic Action for Future CPS Through Roadmaps, Impact Multiplication and Constituency Building* (Road2CPS) project[10] indicated that modelling and simulation is a key research priority for CPS engineering (de Lama & Sinclair 2017).

A model-based approach to CPS engineering is uniquely challenged by the heterogeneity inherent to CPSs. Whilst software engineers intrinsically use rich discrete-event models to describe the control of such systems, it is more appropriate for engineers of physical disciplines to utilise continuous-time techniques to describe the controlled components in the physical world. Fitzgerald et al. (2015) identify that in order to overcome the complexities of the heterogeneity of CPSs, the foundations, methods and tools used in CPS engineering should incorporate both discrete models of computing and the continuous-time models more commonly utilised throughout physical engineering disciplines.

1.2.2 Systems of Systems

SoSs describe an integration of a finite number of independent constituent systems. These constituent systems are networked in order to achieve a goal which is unobtainable by any number of constituents acting alone (Jamshidi 2009*a*). An early definition by Kotov (1997) describes SoSs as "large-scale concurrent and distributed systems, where system components are themselves complex systems".

More recent descriptions conceptualise SoSs as a meta-systems, comprising a number of autonomous constituent systems which might be diverse in technology, context, operation, and geography (Keating

[8]See `https://www.cps-vo.org/`
[9]See `https://ec.europa.eu/programmes/horizon2020/`
[10]See `http://www.road2cps.eu/`

et al. 2003). Popper et al. (2004) describe SoSs as “a collection of task-oriented or dedicated systems that pool their resources and capabilities together to obtain a new, more complex ‘meta-system’ which offers more functionality and performance than simply the sum of the constituent systems”.

Definitions of SoSs vary widely accross application areas; around 40 definitions were collated by Boardman et al. (2006), including publications from industry, government and academia. Whilst no single definition is given, there are a number of themes common accross domains. SoSs typically describe a collection of task-oriented or dedicated constituent systems, where the combination of constituents collectively offer a system with greater functionality and performance than simply the sum of that of the constituent systems, by combining resources and individual capabilities in a collaborative fashion (Jamshidi 2009*b*). INCOSE define SoSs as “an interoperating collection of component systems that produce results unachievable by the individual systems alone”.

The most widely used description of SoSs comes from Maier (1998), who proposed five characteristics which distinguish SoSs:

Operational independence describes how each constituent system serves a purpose outside of the SoS, and can operate indpendently of the SoS.

Managerial independence describes that each constituent system might be controlled under the guidence of different authorities, where each authority has constraints and goals outside of the SoS.

Emergent behavior describes how an SoS might be expected to exhibit behaviours that cannot be achieved by any of the constituent systems operating independently. An emergent behaviour cannot be predicted based on a knowledge of the constituent systems.

Evolutionary development describes how an SoS might change over time, through the addition or removal of constituents or aspects of their functionality.

Geographically distributed describes that constituent systems are unlikely to share the same physical environment.

Of these five, Maier (1998) identifies operational and managerial independence as the two principal distinguishing characteristics for applying the term “system-of-systems”, and claims that systems to which these two characteristics do not apply should not be considered a SoS, regardless of the complexity or geographic distribution of its components. Furthermore, he argues that complexity and geographic distribution of constituents are not necessarily good discriminators for an SoS taxonomy, though they are often associated with the SoS concept. Subsequent definitions also adopt the general concepts of independence, continuous evolution and emergence, e.g. Fisher (2006), Abbott (2006).

Boardman & Sauser (2006) additionally describe how SoS constituents are often heterogeneous, with a typical SoS combining systems from complementary domains. The notion of constituent heterogeneity is also described by Baldwin & Sauser (2009).

Further to the characterisation of SoSs, broad categories of SoS have been proposed to reflect variations in the hierarchy of constituent systems, as well as the intended purpose of constituents and the nature of their governing bodies (Maier 1998):

Directed SoSs have a specific and predefined purpose. Whilst constituent systems in a directed SoS are able to operate independently, their priority is to ensure the intended behaviour of the SoS.

Acknowledged SoSs have agreed objectives, management and resources. Constituent systems in an acknowledged SoS still maintain their own objectives, management and resources alongside those of the SoS in which they participate.

Collaborative SoSs result from voluntary interaction of constituent systems. The constituents of a collaborative SoS collectively agree upon the provision of services.

Virtual SoSs have no central management or agreed purpose. Constituent systems in a virtual SoS are often unaware of their participation in the SoS.

SoS engineering is hindered by a range of challenges associated with the independence and heterogeneity of constituent systems, as well as their tendency to evolve. Where constituents are independently owned and managed, it is common for there to be a lack of disclosure between them. Where constituents are closely integrated, identifying system boundaries is not trivial. It is typical for SoSs to have long life-cycles, and a requirement to include legacy components. Within these long life-cycles, constituents often change or even withdraw from the SoS, without consideration of the impact on the SoS. Furthermore, verification of all possible emergent behaviours of an SoS is particularly challenging.

In an effort to overcome the technical, organisational, political, and sociological challenges of SoS engineering, the European Commission has funded a cluster of research projects (Thompson 2013*b*). The *Roadmaps for System-of-Systems Engineering* (Road2SoS) project[11] indicated that key technical challenges for SoS engineering include modelling and simulation and architectural patterns (Albrecht & Reimann 2013).

DeLaurentis (2005) presents a necessity for SoS-level modelling, describing SoS problems as a collec-

[11]See `http://road2sos-project.eu/`

tion of trans-domain networks of heterogeneous systems that are likely to exhibit emergent and evolutionary behaviours, and argues that such behaviours cannot be explored if the constituent systems and their interactions are modelled separately.

1.3 Research Challenges

Foundations, methods, and tools for engineering both CPSs and SoSs have been developed extensively in recent years, with significant research funding invested in both Europe and the United States. As discussed in Section 1.2, there is significant support for the utilisation of models for overcoming the technical challenges associated with the development of such systems.

A key challenge for employing a model-based approach to CPS engineering stems from their inherent heterogeneity. Whilst techniques for harnessing both discrete-event and continuous-time environments in a single model have advanced, their emergence from the domain of embedded systems has led to the use of models which largely constrain system abstractions to stand-alone entities. Findings from the *Trans-Atlantic Modelling and Simulation for Cyber-Physical Systems* (TAMS4CPS) project[12] suggest that "CPS may operate as individual systems, but are more usually networked as SoS displaying complex behaviours that cannot be adequately predicted with current modelling capabilities" (Hafner-Zimmermann & Henshaw 2017).

Effective engineering of SoSs can also be facilitated by utilising a model-based approach. Organising the hierarchical complexity typical of such systems can be aided with the use of architectural models, guided by the provision of patterns and frameworks. Additionally, exploration of emergent behaviours can be achieved by employing appropriate simulation techniques (Albrecht & Reimann 2013). An array of tools and techniques have been developed to enable the use of models for SoS engineering, but they are largely constrained to the expression of discrete-event phenomena, and offer little provision for detailed expressions of the physical world. Advancing technology affords modern systems both increased computational and communication abilities, leading to a necessity for SoS engineering to facilitate high volume information exchange between large networks of both sensing and actuating devices.

As more devices become 'smart', embedded computers become both more complex and more interconnected. Tomorrow's energy infrastructure, transportation networks, manufacturing systems and buildings will comprise a global network of smart elements. These infrastructures constitute CPSoSs. CPSoSs comprise an array of both physical devices and computing elements which are connected both

[12]See `http://www.tams4cps.eu/`

physically – by flows of energy or material – or virtually – by highly dynamic flows of information. Owing to their ability to generate and communicate large volumes of information, advantages of CPSoSs include improved energy and resource efficiency, and the provision of better services and products in a cost effective and sustainable way. However, the complexity of their interconnectivity makes CPSoSs uniquely challenging to engineer.

Engell, Paulen, Reniers, Sonntag & Thompson (2015) distinguishes CPSoSs as "CPS that exhibit the features of SoS", and summarises their key characteristics:

Size and Distribution. CPSoSs are typically large, often spatially distributed physical systems with complex dynamics. The performance of the CPSoS relies on the orchestration of its constituents, which might be geographically distributed over a large area (e.g. national infrastructure) or be locally concentrated (e.g. a smart building).

Management and Control. CPSoSs cannot be controlled and managed using a centralised or hierarchical top-down approach because of the scope and complexity of the ownership and management structure of its constituents. CPSoS constituents are not controlled or managed by a single authority, but authority is distributed to autonomous constituents, where both local and global control is determined by technical, economic, social and ecologic considerations.

Constituent Autonomy. It is common for CPSoS constituents to pursue local goals independent of the objectives of the CPSoS. Whilst constituents collaborate to provide a particular service, they may also offer desirable services independently. Since constituents are not controlled centrally, the CPSoS must provide incentives or constraints to ensure its autonomous constituents also contribute to achieving global objectives.

Dynamic Reconfiguration. Dynamic reconfiguration of the overall system is typical for CPSoSs. The addition, modification, and removal of constituents is common for CPSoSs, where constituents might be transient (e.g. aircraft in air traffic control), or alter their structure or management strategies following changes in resource availability, demands or regulations.

Continuous Evolution. Typically operational for long periods, CPSoS are continuously improved throughout their life-cycle. Whilst computing infrastructure and communication architectures are replaced or updated frequently, physical hardware and specialist software is more commonly in operation for decades, and additional functionality or performance improvements must be implemented with limited modification of some parts of the overall system.

Emergent Behaviour. The overall behaviour of a CPSoS results from interactions between its constituents. Constituents might interact by the exchange of digital information, or by some physical connections. Often considered problematic because of their unpredictability, emergent behaviours enable the design and management of CPSoSs without detailed knowledge of their constituents.

The design and management of CPSoSs cannot be based on theories and tools from any single domain, but instead requires a multidisciplinary approach (Lee 2008, Broy 2013). The behaviour of physical elements must be modelled, simulated, and analysed using methods from continuous systems theory, such as large-scale simulation and stability analysis. Additionally, methods and tools from computer science should be employed for the modelling of distributed discrete systems, to enable verification of both low-level and global behaviours based on abstract descriptions (Engell, Paulen, Reniers, Sonntag & Thompson 2015).

Thompson et al. (2015) argue that CPSoSs are of particular importance because they represent some of the most important infrastructures, including the generation and distribution of energy, drinking water, rail, road, air and marine transportation, and large industrial production processes. Both CPS and SoS research also emphasise the importance of the emergence of CPSoS, which has led to investment in identifying and overcoming challenges particular to this unique class of system.

European investment in CPSoS research includes the *Towards a European Roadmap on Research and Innovation in Engineering and Management of Cyber-Physical Systems of Systems* (CPSoS) project[13], intended as an exchange platform for SoS related projects and communities (Reniers & Engell 2014), with an aim to develop a research and innovation agenda on CPSoS based on experience from both academia and industry (Engell, Paulen, Reniers, Sonntag & Thompson 2015). Based on consultation with domain experts, outputs from the CPSoS project include a summary of the state of the art and future challenges for CPSoS (Thompson et al. 2015), and a research and innovation agenda (Engell, Paulen, Sonntag, Thompson, Reniers, Klessova & Copigneaux 2015).

Investigation into a range of application domains, including transport, energy infrastructure, manufacturing, and buildings, led to the identification of three core long-term research challenges for the design and operation of CPSoSs:

- Enabling distributed, reliable, and efficient system management
- Providing engineering support methodologies and software tools

[13]See `http://www.cpsos.eu/`

- Facilitation of cognitive features to aid human operators of highly complex systems

These challenges were identified by the CPSoS consortium based on consultation with around 180 practitioners and experts from key organisations around the world (Thompson et al. 2015). This consultation revealed a collection of concerns which must be addressed by future research in order to overcome the core challenges.

In order to enable distributed, reliable, and efficient system management, future research must identify the impact of management structures (e.g. centralised/distributed) on system performance and robustness. Since partial autonomy of constituents leads to uncertain behaviours, stochastic techniques for optimisation and risk management are required to provide assurances about the performance of the overall system. Furthermore, distributed management and control methods are required which prevent dynamic interactions between autonomous constituents resulting in the emergence of undesirable behaviours.

Typically socio-technical systems, additional uncertainty is introduced to CPSoSs by the inclusion of human operators and managers. Research is required to establish techniques for anticipating human behaviours and for understanding how to optimally combine the capabilities of humans and machines in real-time monitoring and decision making. To make CPSoSs resilient to the impact of the naturally unpredictable behaviour inherent to humans, user behaviour and its consequences on the system must be continuously analysed. Decision support tools are required to assist system operators, where a capacity for self-learning can enable the identification of optimal operational patterns from past examples.

Because it is common for CPSoSs to be operated and continuously improved over long time-scales, much of their engineering must be undertaken at runtime. The provision of engineering support methodologies and software tools must include techniques to facilitate the addition and modification of components already deployed and interacting with other constituents. The overlapping of design, engineering, and operational phases necessitates engineering frameworks which support the specification of requirements, structural and behavioural modelling, and system realisation throughout the complete system life-cycle. In order to establish, validate, and verify key properties of CPSoSs, algorithms and tools are required which enable automated analysis of complete, large-scale, dynamically evolving systems.

Enabling modelling and simulation of CPSoSs requires tools and techniques for model management, and for the integration of diverse models from different domains. Effective model-management requires meta-models which support the automation of model transformations, and analysis of global properties of CPSoSs requires high-level models and efficient simulation algorithms which enable system-wide simulation of collections of heterogeneous constituents. Model-based development of CPSoSs demands collaborative environments which support the integration of models from competing stakeholders and legacy components.

Because of their scale and complexity, failures are commonplace in CPSoSs(Thompson et al. 2015, p. 18). Error detection and fault tolerance mechanisms are required which handle the propagation of faults over different layers of system hierarchy. Research is also required to understand cyber security concerns particular to CPSoSs. Unique challenges include the recognition of malicious injections and constituent hijacking, where detection of such attacks requires consideration of both the physical and computational elements.

To operate CPSoSs both efficiently and robustly, there is a requirement to detect operational changes in demand and to handle system anomalies and failures. Gaining this situational awareness requires large-scale data acquisition for optimisation, decision support, and control. By gathering large volumes of data, support is needed for the integration of complex data acquisition systems, and the management of data collected from a variety of sources. Substantial processing capabilities (e.g. cloud-based computing) are required for the real-time analysis of data to monitor system performance and detect faults or degradation, and novel visualisation tools and techniques are necessary to manage the complexity of the data to enable effective risk management and decision support.

The proposed research aims to address a subset of these challenges, in particular those related to the support of MBSE for CPSoS. Thompson et al. (2015) summarise that effective engineering of CPSoSs requires a model-based approach which enables the description of system configuration at the architectural level, as well as simulation of heterogeneous complex systems. They stipulate that these complementary models should be linked to ensure semantic consistency. It is addressing these concerns in particular with which the proposed research is concerned.

1.4 Proposed Research

Muller (2013) proposes that systems engineering research should outline objectives to overcome some weakness in the state of the art, and that the effectiveness of the proposed solution should be evaluated against some criteria. This section summarises the particular weaknesses that the proposed research aims to address and introduces a set of objectives which should be met in order to overcome them. The proposed approach to undertaking the research is described in Section 1.4.1 and the research contributions made by the completion of the work are detailed in Section 1.4.2.

A great deal of progress has been made in developing tools and techniques for constructing models in such a way as to preserve the heterogeneity inherent to CPS by harnessing both discrete-event and continuous-time environments in a single model. Originating from embedded systems research, this approach generally considers a CPS as a stand-alone entity, with provision for close interaction across the cyber-physical boundary, but little consideration of interaction between a number of independently owned and managed systems.

SoS research has developed a number of techniques for managing the complexity of SoS architectures. By the provision of architectural patterns and frameworks, detailed descriptions of constituent interconnectivity can be expressed, but such expressions are typically employed for the description of information exchange between digital systems.

The state of the art of MBSE of CPSoSs is limited in its absence of an integrated approach which includes consideration of both the organisational requirements of characteristics common to SoSs, and the modelling and simulation challenges associated with the heterogeneity of CPS. Building on techniques established separately for CPS and SoS engineering, we conjecture that:

> *By employing modelling technologies for both CPSs and SoSs, it is possible to support activities concerning the engineering of CPSoSs, where the overall behaviour of such systems is a result of a close coupling of software and physics, and interaction between a number of independently owned and managed systems.*

In evaluating the effectiveness of an approach to aid the engineering of CPSoSs, it is important to consider the specific engineering processes that the proposed techniques will support. In Section 1.3 we specify that the proposed research is concerned with the use of models to enable the description of system configuration at the architectural level, as well as facilitating simulation of heterogeneous systems. The use of models in the description of system architectures leads the proposed techniques to support engineers in the architecture definition process. Furthermore, by the provision of a systematic approach to the description of system architectures, the proposed techniques will also support engineers in undertaking the design definition process. Finally, by utilising more concrete models to support simulation of system behaviours, the proposed approach also explicitly supports the system analysis process.

It is also important to consider the characteristics of systems to which use of the proposed techniques will be advantageous. Since the approach primarily aids engineering activities in the early stages of the system life-cycle, it will be broadly limited to CPSoSs which are in some way designed. This excludes, for example, virtual SoSs, where emergent constituent interaction is observed in operation but not necessarily intended by design. The more open and collaborative the constituent system owners and designers, the more lucrative the proposed approach will be. Additionally, the proposed approach is intended to aid a particular system design or configuration, in a relatively static way. Highly reconfigurable systems would gain relatively little reward from the utilisation of such an approach.

In advancing the state of the art in MBSE for CPSoS, the proposed research must satisfy a series of objectives. Evaluation of the success of the approach can be determined by consideration of how effectively each objective is met by the proposed techniques. The primary objectives of the research are:

Facilitate the definition of architectural descriptions of a system of interest.
Architectural descriptions enable the detailed specification of constituent structures and behaviours, as well as their interfaces and interactions with other constituents. The production of architectural descriptions must be guided in such a way to ensure that these specifications are sufficiently detailed to enable the production of more concrete models and implementations. Specifically, guidance is required which ensures that architectural descriptions can be expressed with sufficient detail to support the production of an executable heterogeneous model.

Enable the realisation of architectural descriptions.
Further to the provision of design aids for ensuring sufficient detail in an architectural description, there should be guidance for the realisation of architectural descriptions using an appropriate architectural description language. An appropriate formalism should be tailored to support the description of CPSoS architectures, including the specification of discrete event phenomena and continuous time phenomena from a diverse range of engineering domains.

Support the production of executable models based on architectural descriptions.
Whilst architectural models enable organisation of constituents and their interfaces, they lack any provision to explore the impact of design decisions on behaviours which emerge as a result of interaction between constituents, and between cyber and physical elements. To achieve this, more concrete models are required which enable the simulation of CPSoS behaviour including both discrete event and continuous time elements. The production of these models should be guided in such a way that ensures the

semantics of complementary abstractions of a system are consistent.

Evaluate the effectiveness of the proposed techniques.
Verification of the proposed techniques should be undertaken by employing them in the description of an example system of interest. Application of the approach to an archetypal study not only demonstrates the approach, but also provides a basis for its evaluation. Furthermore, the approach should be validated against a suitable validation framework.

1.4.1 Approach Overview

In order to satisfy the research objectives, the proposed approach aims to utilise a combination of existing technologies in a systematic way.

Firstly, the approach must facilitate the production of high-level architectural descriptions of CPSoSs, to enable the specification of important properties of any constituent systems, and any relationships between them. These models must include descriptions of constituents in terms of cyber and physical elements and any interfaces between them, and be sufficiently detailed such that they facilitate the production of a less abstract models, capable of simulation. In order to ensure this richness of architectural descriptions, the proposed approach supports their production by employing an appropriate architectural framework, specifically intended to aid the development of CPSoSs.

Whilst an architectural framework ensures that an architectural description is sufficiently detailed for its intended purpose, it does not specify a particular notation for realising the architectural description. An appropriate systems modelling language is necessary for the provision of architectural descriptions. The chosen language must be domain independent to support the description of phenomena from a variety of disciplines, and include syntax for the expression of both cyber and physical phenomena, as well as constituent properties and interfaces. The proposed approach includes guidance for the use of an appropriate language for realising architectural descriptions by the provision of a systems modelling language profile.

By supporting the production of architectural descriptions, the proposed approach will enable detailed descriptions of CPSoSs, however these abstract models are limited. Complex, emergent behaviours, determined by the interaction between constituents composed of both cyber and physical elements demands a simulation-based modelling environment. These executable heterogeneous models (‘*co-models*) must facilitate discrete-event descriptions of digital phenomena, as well as continuous-time descriptions of physics. With complementary abstractions of the same CPSoS, the proposed approach must ensure semantic consistency between descriptions. To achieve this, the production of co-models should be guided in such a way as to preserve any details included in the architectural description. The proposed approach supports the automation of model refinement by the specification of a mapping between information contained within an architectural description and the data structure of executable models.

The proposed approach supports the engineering of CPSoSs by the provision of a collection of mechanisms to support the development of models. These mechanisms support the development of two distinct abstractions of a system of interest: An architectural description and an executable co-model. To aid the development of an architectural description, the approach includes an architectural framework and a supporting language profile for its realisation using a suitable ADL. To aid the development of co-models, the approach includes a specification for a mapping from architectural descriptions to co-models, facilitated by the abstract representation of the underlying meta-models. The facets of the proposed approach are summarised in Figure 1.1.

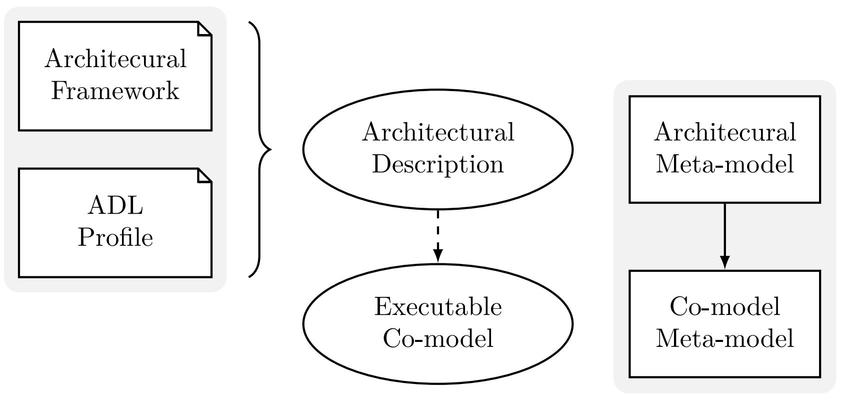


Figure 1.1: MBSE for CPSoS—Facets of a Systematic Approach

1.4.2 Research Contributions

The proposed research utilises existing technologies in a unique and systematic way to support a model-based approach to engineering CPSoSs.

By investigating the suitability of candidate technologies, this work provides a summary of the state of the art in model-based engineering techniques, with consideration of the appropriateness of each technology for CPSoS engineering.

The research includes the provision of a bespoke architectural framework, a supporting language profile for realising architectural descriptions, and a mapping between abstract representations of architectural and executable models.

Application of the proposed techniques enables assessment of its effectiveness, and by consideration of the impact of the approach on the suitability and maturity of baseline technologies for their use in CPSoS engineering, the utility of the techniques are evaluated.

1.5 Document Outline

The remainder of this thesis provides a critical evaluation of the current state of the art in model-based CPS and SoS engineering, followed by a description of the proposed approach to modelling CPSoSs. The approach is demonstrated using an example study, and the utility of the approach is evaluated. The strengths and weaknesses of the approach are considered, alongside promising opportunities for further work.

Chapter 2 explores existing work in relevant domains, surveying the state of the art in related fields of study. With the state of the art identified, limitations of the current literature are discussed, and the contributions of this thesis are distinguished from existing work.

Chapter 3 provides an overview of the various research activities necessary for achieving the research objectives. The selection of appropriate research methods for undertaking each activity is described, where method selection is informed by consideration of any methodological implications.

Chapter 4 describes the development of an architectural framework for the production of architectural descriptions of CPSoSs. The architectural framework provides a requisite set of viewpoints of a system of interest, necessary for the development of a comprehensive model of the system. This is supported by the definition of a language profile, which constrains the use of a domain independent systems modelling language specifically for the realisation of the viewpoints of the architectural framework.

With the tools for developing architectural descriptions established, Chapter 5 presents a specification for the mapping of architectural models to executable co-models. An abstract representation of the meta-models underpinning each of the candidate modelling formalisms is developed, and used in the definition of a transformation process for translating between complementary abstractions.

Chapter 6 presents a case study, where the application of the proposed techniques to an example CPSoS is used to verify the approach. The case study describes the development of models for the description of a rail interlocking scenario. Simulation of a co-model based on an architectural description of the system enables exploration of the impact of control strategies on safety, network capacity, and energy use.

In Chapter 7, mechanisms for the assessment of the utility of the approach are described, and applied to the proposed techniques. By consideration of the impact of the approach on the suitability and maturity of baseline technologies, the effectiveness of the proposed techniques is evaluated.

Finally, Chapter 8 considers the strengths of the approach, its limitations, and possible opportunities for its further development.

1.6 Summary

CPSoSs describe continuously evolving, complex systems, subject to distributed management, both autonomous and human control, and dynamic reconfiguration. Their particular combination of characteristics makes CPSoSs uniquely challenging to engineer.

Model-based techniques can be employed in overcoming engineering challenges, however the utility of existing technologies is limited for the engineering of CPSoSs. A systematic approach to the utilisation of existing model-based techniques is proposed, which combines complementary approaches to overcome the challenges of CPSoSs.

The proposed approach supports the definition of architectural descriptions by the provision of an architectural framework and supporting profile to tailor an architectural description language for the realisation of the framework. Furthermore, by the specification of a translation between architectural and less abstract executable models, the approach enables the simulation of models to facilitate analysis of emergent properties and behaviours.

Techniques for the evaluation of the approach are described, including the development of a case study describing a rail interlocking system, and a framework for assessment of the utility of the approach.

Related Work 2

Chapter 1 identifies characteristics of CPSoSs by separate consideration of the characteristics of CPSs and SoSs. Key challenges are associated with the engineering of such systems, and a variety of approaches for overcoming these challenges have been established. The proposed research aims to systematically utilise a combination of existing CPS and SoS engineering approaches to aid the development of CPSoSs.

This chapter introduces existing tools and techniques which might be employed in a model-based approach to systems engineering. Section 2.1 introduces a diverse collection of model-based techniques which might be used to support a range of engineering activates.

Section 2.2 summarises the state of the art in CPS engineering, and Section 2.3 introduces the state of the art in SoS engineering. Each of these sections first introduces a broad overview of engineering approaches, followed by an evaluation of existing model-based techniques. Consideration is given to the limitations of existing technologies for CPS and SoS engineering when used to support the development of CPSoSs.

2.1 Models and Modelling

Models provide a simplification of reality and enable better understanding of a particular system or phenomena. Model-based engineering utilises models to support the design (Sztipanovits & Karsai 1997) and development (Selic 2003) of systems, where they provide specifications for systems and reflect the evolution of the system design. Models can be used to visualise a system, specify the structure or behaviour of a system, provide a template for the development of a system, document the design of a system, and support automated synthesis of system implementations (Rumbaugh et al. 2004, van

Amerongen 2010). The use of models can enable various analyses and facilitate simulation of system behaviours, aiding early identification of design weaknesses.

A wide range of concepts or artefacts might be considered models (Rumbaugh et al. 2004, Holt & Perry 2008). A model can be textual (e.g. written specifications), and use natural language to explain a system. Visual models (e.g. blueprints) can provide a template for the creation of a system or the basis of its analysis, whilst physical models (e.g. mock-ups) can illustrate a system under design, or support physical simulation or analysis. Mathematical models range from simple equations to complex formal specifications, and can enable sophisticated reasoning about a system. In selecting an appropriate approach to modelling, Rumbaugh et al. (2004) propose four basic principles of modelling:

1. An approach to overcoming a problem and the resulting solution is significantly influenced by the selection of particular models. Many types of model might be employed for a given application, but it is important that appropriate models are identified.

2. Any model might be expressed at different levels of abstraction. Whatever the chosen modelling formalism, it is important that it is flexible in facilitating representation of a system of interest with varying levels of precision depending on the requirements of the application.

3. Effective models are well connected to reality and the value of a model is limited to how well it represents the system being modelled. Whilst models facilitate a simplification of reality, it is important that any simplifications do not obscure important details.

4. No single model is sufficient for the simplification of a system. For the representation of any nontrivial system, it is important to consider a selection of independent but related models.

These principles should be considered when determining an appropriate combination of modelling technologies for supporting the development of CPSoSs. Common to all approaches is the expression of system elements using an appropriate formalism or language, where a particular syntax is employed for the representation of artefacts.

Modelling Languages

The expression of a model in a particular formalism is constrained by a set of rules which determine how information is recorded, and enable interpretation of the meaning of elements in a model. These rules are described as a *modelling language*, and modelling languages are commonly used throughout computer science, information management, software and systems engineering, and business process modelling to specify system requirements, structures, and behaviours.

Modelling languages might be graphical or textual (He et al. 2007). Graphical modelling languages are underpinned by the creation of diagrams; they specify named symbols to represent concepts, and connectors to represent relationships between concepts. Additional symbols are often used to represent properties of a concept or relationship or constraints over them. Textual modelling languages typically specify a set of keywords which are coupled with parameters or expressions to create meaningful representations of the information.

Modelling languages enable precise and consistent specification of systems. Basic modelling languages are often limited to the provision of representations of system properties and characteristics. More sophisticated modelling languages are often executable, and facilitate automation of system verification, validation, and simulation, and the synthesis of more concrete artefacts (e.g. code generation). These languages typically have a formal specification for their use (Reghizzi et al. 2013) and are often supported by enabling tools.

Modelling languages might be *general purpose* or *domain specific*. General purpose modelling languages have a wide variety of purposes across a broad range of domains. To achieve this, they support low-level abstractions, and often include mechanisms to extend or specialise the language for a particular application. General purpose modelling languages typically focus on concepts the acquisition, sharing, and utilisation of knowledge, and are most commonly graphical languages (Mazanec & Macek 2012). By far the most widely used general purpose modelling language is the Unified Modelling Language (UML; OMG 2015*b*). UML is a graphical language and defines 13 types of diagram to facilitate the specification of requirements, structure, and behaviour of a system. It includes a profile mechanism which enables it to be tailored for use in specific domains using stereotypes, stereotype attributes, and constraints, to both restrict and extended the scope of the language. UML was originally intended for the development of software systems but is often used for systems engineering and process modelling. This extended use of UML led to the development of the Systems Modelling Language (SysML; OMG 2015*a*). SysML is a UML profile which removes or generalises elements specific to software engineering and introduces features to tailor its use for systems engineering (Holt & Perry 2008).

In contrast to general purpose modelling languages, domain specific modelling languages typically support higher-level abstractions, and require fewer details to specify a particular system. Domain specific modelling languages target a specific, concrete domain, and so their use is often very limited (Fowler 2010). They are often textual languages, with more expressive and less ambiguous syntax than graphical alternatives (Mazanec & Macek 2012), and in many cases these languages are formally defined.

Examples of domain specific languages are introduced in sections 2.1.1–2.1.3.

2.1.1 Requirements Elicitation

In systems engineering, models can be used in the specification and analysis of system requirements. Requirements engineering includes the definition, documentation, and maintenance of system requirements (Kotonya & Sommerville 1998, Nuseibeh & Easterbrook 2000), and depending on the type of system being developed, can benefit from the undertaking of a range of activities, supported by a variety of models (Sommerville 2007).

To enable the specification, modelling, and analysis of requirements, they must first be established through a process of elicitation. Requirements elicitation describes the identification of requirements through engagement with key stakeholders, and typically involves a series of meetings or interviews (Sommerville & Sawyer 1997). The identification of requirements can be guided by a complementary set of approaches, including the utilisation of a range of models (Alexander & Beus-Dukic 2009).

Requirement elicitation might be aided by the modelling of system objectives, or *goal modelling*. A goal describes an objective to be achieved by a system through cooperation of actors in the system environment (Liu & Yu 2004). A goal model expresses the relationship between a system and its environment, clarifies system requirements, enables requirement conflict management and requirement completeness measurement, and explicitly connects requirements to elements in the design process. Graphical modelling languages are typically used to implement goal models, such as the Goal-oriented Requirements Language (GRL; Yu et al. 2011), which is designed to support the development of goal-oriented models and reasoning about requirements. Similarly, Knowledge Acquisition in automated Specification (KAOS; Lapouchnian 2006) is a goal modelling language underpinned by formal methods of analysis. General purpose languages might also be used for modelling goals, such as UML Use Case Diagrams (Alexander & Beus-Dukic 2009, p. 121).

Once determined, requirements are documented in a system requirements specification, which might also be implemented with the aid of textual and graphical modelling notations. A robust specification of system requirements aids the design of the structure and behaviour of a system.

2.1.2 Describing Architecture

Both the structure and behaviour of a system are described by a *system architecture*. A system architecture includes descriptions of the major components of a system and any relationships and interactions between them (Jaakkola & Thalheim 2010). It should consider development processes and be flexible to evolve and reflect changes in a system over time (Sanders & Curran 1994). The ISO define a system architecture to be the “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution” (ISO/IEC/IEEE 42010 2011).

The development of a system architecture is described as *architecting*. Architecting is undertaken within the context of a project and/or organisation, and is performed throughout the entire life cycle of a given system. It is defined by the ISO as “the process of conceiving, defining, expressing, documenting, communicating, certifying proper implementation of, maintaining and improving an architecture throughout a systems life cycle”. System architecting can be supported by the use of models, in particular by the use of Architecture Description Languages (ADLs; Clements 1996, Medvidovic & Taylor 2000). ADLs are used throughout systems and software engineering, and are described by the ISO as “any form of expression for use in architecture description”.

Using an ADL for the representation of architectures supports effective communication between stakeholders, provides a mechanism for recording early design decisions, and facilitates the creation of transferable abstractions of a system. ADLs can be graphical, textual, or a combination of both. Basic graphical representations of system architectures provide useful documentation, but their lack of formality limits their usefulness (Allen & Garlan 1997, Perry & Wolf 1992). More sophisticated ADLs are defined formally to overcome these limitations; they support analysis of a system early in its development and facilitate feasibility testing of design decisions.

A wide variety of ADLs are available, offering a range of conceptual architectural elements for diverse applications. Many ADLs target engineering software architectures, for communicating architecture to both software developers and users. Notable examples include Acme, a simple language with concepts of systems, components, connectors, ports, roles, and representations, where a system is composed of components which are related by connectors (Garlan et al. 1997), and Darwin, which supports hierarchical composition and software parallelism (Magee et al. 1995).

ADLs have been proposed which focus on a particular operational domain, or are targeted at particular methods of analysis. Examples of ADLs targeting a particular domain include those intended to aid the development of real-time and embedded systems, such as the Architecture Analysis and Design Language (AADL; Feiler et al. 2006), and the Embedded Architecture Description Language (EADL; Li et al. 2010). ADLs intended to support particular types of analysis have been proposed for the analysis of system availability, reliability, security, resource consumption and trustworthiness, such as the Trustworthy Architecture Description Language (TADL; Mohammad & Alagar 2008).

Despite the availability of a broad range of ADLs, they are seldom used for describing software in industrial practice. Suggestions of the cause of their unpopularity include poor tool support and documentation, and a lack of generalisability and extensibility (Woods & Hilliard 2005, Pandey 2010, Clements 1996). In overcoming these limitations, UML has been proposed as a more appropriate approach to modelling software architecture (Ivers et al. 2004, Pérez-Martínez & Sierra-Alonso 2004).

ADLs have also been proposed to support the description of enterprise architectures. The Archimate ADL does not refer to software components, but includes components to support the description, analysis and visualisation of architecture across business domains (Lankhorst et al. 2009).

Architectural Frameworks

The development of an architecture description for a particular application can be guided by the use on an *architectural framework*. An architectural framework provides principles and practices for the creation and utilisation of an architectural description of a system (Holt & Perry 2008). The ISO define an architectural framework as "conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholder" (ISO/IEC/IEEE 42010 2011).

An architectural framework typically specifies a complementary set of *viewpoints* for describing an architecture, enabling a systematic approach to the architectural design. A viewpoint provides a specification for the implementation of a particular *view* of a system, which is implemented using an appropriate ADL. Each view of a system offers a complementary perspective of the system for a particular purpose. Whilst an architectural framework specifies any number of viewpoints deemed necessary for the specification of an architectural description, it does not necessarily direct how the views should be derived (Holt & Perry 2008).

There is a broad range of popular architectural frameworks used throughout academia and industry, each focussed on the development of particular types of system. Holt & Perry (2010) propose that an architectural framework is essential for the development of a robust architectural description, but is it is important that an appropriate framework is used. Popular architectural frameworks include The Open Group Architectural Framework (TOGAF; Haren 2011) and the Zachman framework (Zachman 2008) for the developemnt of enterprise architectures. Defence-based frameworks such as the UK Ministry of Defence Architectural Framework (MODAF; Biggs 2005) and the US Department of Defence Architectural Framework (DoDAF; US Department of Defense 2003) are also widely used, but are intended to support system acquisition.

2.1.3 Formal Methods

The utilisation of mathematical models for the analysis and verification of a system (particularly in computing) includes *formal methods*. Formal methods are mathematical techniques for the development of software and hardware systems, where mathematical rigour facilitates the analysis and verification of models to support engineering activities throughout the life-cycle of a system (Woodcock et al. 2009).

Formal methods can be utilised in the elicitation, articulation, and representation of requirements (George & Vaughn 2003), and tools can provide automated support for checking completeness, traceability, verifiability, and reusability. Additionally, formal methods can support the evolution of requirements, viewpoint diversity, and the management of inconsistency (Ghose 2000).

The specification of software can also benefit from the application of formal methods, facilitating an exact statement of what software should do without any constraint of how it should be achieved. Such a specification facilitates a universal understanding of the purpose of the software between technical and non-technical stakeholders. Examples of such methods include Abstract State Machines (ASM; Börger & Stärk 2003), B (Abrial 1996), and the Vienna Development Method (VDM; Jones 1990).

Formal methods can be used in the organisation of the architecture of complex software systems. Models which supress details of a particular implementation enable system architects to focus on analysis and design decisions important to structuring the system in such a way as to satisfy its requirements (Allen & Garlan 1992, Van Lamsweerde 2003). The Wright ADL provides a formal basis for architectural descriptions, and can be used to provide a precise, abstract meaning to an architectural specification (Allen 1997).

Formal methods can be challenging to apply, but advances in their automation have increased their viability, particularly in software development (Woodcock et al. 2009).

2.1.4 Simulation

Complementary to static approaches to the representation of systems, are models which enable *simulation*. Simulation is described by Robinson (2014, p. 5) as 'experimentation with a simplified imitation of a system as it progresses through time, for the purpose of better understanding and/or improving that system', and that it is a 'what-if analysis tool'.

Simulation is often a particularly expensive approach to modelling, and is recommended by Pidd (2004) to be used as a last resort, rather than the preferred choice. Whilst there are many scenarios where alternative modelling approaches are appropriate and simulation is not necessary, there are also scenarios where simulation is the only way to analyse a particular system, and surveys of modelling practice have shown simulation to be one of the most widely used modelling techniques (Jeffrey & Seaton 1995, Jahangirian et al. 2010). The unique merit of simulation models stems from their ability to represent the *variability*, *interconnectedness*, and *complexity* of a system (Robinson 2014).

The variability of a system includes both predictable (e.g. the scheduled availability of a resource) and unpredictable (e.g. system usage) variations, and both are typical of many complex systems. Whilst some other approaches to modelling can be adapted to account for variability, it often significantly increases the complexity of the model, and systems which are subject to substantial variation cannot always be modelled analytically. Where this is the case, simulation is often the only means for accurately predicting the performance of the system. The interconnectedness of a system describes the ways in which the behaviour of system components impacts other components. It is often challenging to forecast the impact of the interconnections in a system, especially when coupled with variability.

Robinson & Higton (1995) demonstrate the limitations of static approaches to modelling variability and interconnectedness by considering alternative factory designs where variability results from equipment failure. In a static analysis, equipment failure was accounted for by factoring its effect into process cycle times, but in simulation models, failures were more easily modelled in more detail. The static analysis indicated that each of the designs was capable of achieving the required throughput, whilst the simulation demonstrated that none of the designs were satisfactory.

The complexity of a system can be described in terms of *combinatorial* and *dynamic* complexity (Brooks & Tobias 1996). Combinatorial complexity describes the number of components, or feasible combinations of components in a system, whilst dynamic complexity arises from the interaction of system components over time (Sterman 2000). Dynamic complexity leads to different consequences of an action in the short and long term, different consequences of an action in different parts of a system, and counter intuitive system behaviours (Senge 1990). These effects lead to significant challenges in predicting the performance of a complex system.

It is typical for nontrivial systems to be subject to variability, interconnectivity, and both combinatorial and dynamic complexity. Each of these characteristics introduces significant challenges to predicting the performance of a system using a static modelling approach, and when combined, can make such approaches unsuitable. In contrast, simulation models facilitate explicit representation of these characteristics and enable the prediction of system performance and the comparison of alternative designs of such systems (Robinson 2014).

Simulation models might be *deterministic* or *stochastic*. Whilst simulation of deterministic models will always produce the same output for a given input, the output of stochastic models will vary with each simulation for a given input. Stochastic simulation is used to describe systems in which events occur probabilistically. Monte Carlo simulation describes an approach to stochastic modelling and is aimed at modelling risk in an environment where an outcome is subject to chance (Winston & Albright 2015). In a Monte Carlo model, the behaviour of a system is considered as a set of distributions representing variables for each source of chance, and the distributions are combined in some way to calculate the outcome. Monte Carlo models are particularly useful for simulating systems with many coupled degrees of freedom, such as fluids, disordered materials, and cellular structures, and for modelling phenomena subject to significant uncertainty such as the financial performance of investment portfolios. Stochastic simulation might also utilise other types of stochastic model, such as Markov chains, which describe a sequence of events in which the probability of each event is dependent on the outcome of previous events (Gagniuc 2017).

There are many different approaches to the development of simulation models, and many tools and techniques associated with each approach. One particularly polarising characteristic of approaches is the representation of time: simulation models might be *Discrete-Event* (DE) or *Continuous Time* (CT) (Pidd 2004, Law et al. 2000).

Discrete-Event Simulation

In a DE simulation model, the behaviour of a system is often described as a sequential series of activities, where the value of each variable is applicable at a distinct point in time, and time itself is represented as a discrete variable. In DE simulation, “only the points in time at which the state of the system changes are represented” (Robinson 2014, p. 22). DE simulation modelling is typically employed throughout the development of many types of system, including digital hardware (Ashenden 2010, Liu 1998, Thomas & Moorby 2008), communication systems (Banks et al. 2004) and embedded software (Chiodo et al. 1994).

DE simulation can be used to assess the performance of a system where its behaviour is the result of the actions and interactions of autonomous entities, or *agents*. Agent-based simulation describes a subclass of DE simulation tailored for the representation of complex, adaptive systems and subject to emerging behaviours (Heath & Hill 2010). The approach was popularised by the Sante Fe Institute through its Swarm software (Minar et al. 1996), which has been used to model biological, physical, and social systems. In an agent-based model, systems are modelled bottom-up as a set of agents, where each agent is described in terms of its individual behaviour. Agents interact with other agents over time, and these behaviours might be deterministic or stochastic. The approach enables simulation of behaviours, patterns, and structures which emerge through interactivity of system components (Macal & North 2010). Agent-based modelling has been used to represent the evolution of cultures (Axelrod 1997, ch. 7), the dynamics of popular opinion (Deffuant et al. 2002), and the spread of disease (Parker & Epstein 2011). It can also be particularly important in aiding decision making (Robertson & Caldart 2009), for example in the restructuring and deregulation of electricity power markets (Macal & North 2005).

DE simulation can be supported by a wide range of tools depending on the application. For basic simulation, generic tools such as spreadsheets might be used (Seila 2002, Greasley 1998), where built-in functions can be utilised in representing system behaviour, but this functionality is prohibitively limited for modelling outside of the simplest of systems. General purpose programming languages can provide additional flexibility and expressiveness in the design of models for DE simulation, and object-orientated languages can be particularly useful for the development of such models, however the development of bespoke simulation mechanisms is costly (Pidd 1992).

Since the 1960s, specialist simulation languages have been developed to more easily facilitate DE simulation. Early simulation languages such as the General Purpose Simulation System (GPSS; Schriber 1974), and SIMULA (Dahl & Nygaard 1966), enable the specification of code-based models and generate results for a given input, but are limited in aiding the understanding of a system for non-experts. In overcoming this, early attempts to animate a simulation (e.g. Amiry 1965), eventually led to both visual and interactive simulations (Hurrion 1976), and the development of the visual-interactive simulation languages such as SEE-WHY (Fiddy et al. 1981). More recent developments in DE simulation languages include improvements in both the functionality and animation capabilities of simulation tools, as well as compatibility with external resources and facilities to distribute computational load (Law et al.

2000).

Tools supporting the development of models using DE simulation languages might be *general purpose* or *application-oriented*. General purpose simulation tools are designed to aid the development of systems from a wide range of domains, whilst application-oriented simulation tools are specialised for the development of a particular type of system. Application-oriented simulation languages and are typically easier to use but are limited in their applicability (Law et al. 2000). Examples include Care Pathway Simulator (Dodds 2005) and MedModel (Harrell & Lange 2001) for modelling healthcare systems, and AutoMod (Rohrer & McGregor 2002) for modelling manufacturing, distribution, and material handling systems.

The majority of modern DE simulation tools are described by Pidd (2004) as ‘visual interactive modelling systems’, and enable a model to be constructed and simulated using a graphical and highly interactive interface. Such software typically includes a library of predefined modelling objects and can be used without programming expertise, although the inclusion of programming languages is often supported for modelling more complex behaviours.

Continuous Time Simulation

The behaviour of many systems is not characterised by discrete changes in state, but the state of the system changes continuously over time. This is typical of systems which include complex physical processes, such as the movement of fluids. Depending on the level of granularity with which these behaviours must be described, it may not be appropriate to represent such systems using a discrete time model. In a CT simulation model, a system is described in terms of continuous change, where the value of a variable might only be applicable for an infinitesimally short period, and time is represented as a continuous variable. In CT models, “the state of the system changes continuously through time” (Robinson 2014, p. 35). CT simulation models use differential equations and iterative numerical integration methods to describe dynamic behaviour, and are typically employed for the development of analogue circuits and physical processes (Liu 1998).

CT simulation is typically used for models of *system dynamics*. System dynamics describes a CT simulation approach which represents a system as a set of *stocks* and *flows*, where stocks describe accumulations of physical matter and flows adjust the levels of stocks (Forrester 1997, Coyle 1996, Sterman 2000). The value of stocks is constantly changed by the action of flows, so requires a continuous representation of time. One of the most widely used tools for the modelling and simulation of system dynamics is Simulink, which provides an interactive graphical environment to support the design, simulation, implementation and testing of systems (Dabney & Harman 2004). It is often used in the development of systems such as communications, controls, signal processing, video processing, and image processing (Fitzgerald et al. 2014*a*). Similarly, the 20-sim tool supports modelling and simulation of system dynamics. 20-sim models can be constructed as equations, block diagrams, bond graphs, or a combination of these, and are widely used for modelling complex multi-domain systems (Duindam et al. 2009) and the development of control systems (Broenink 1999).

Whilst CT simulation models describe behaviour in terms of continuous change, computation simulators are unable to perform calculations in a continuous way. Since computers calculate results in a discrete and sequential way, tools supporting CT simulation must emulate the continuous progression of time. A CT simulator “approximates continuous change by taking small discrete-time steps (Robinson 2014, p. 35). The resolution of the approximation of time is dependent on the size of the steps taken by the simulator, and since a smaller time step results in an increase in the number of calculations to be made to complete a simulation, there is a trade-off between the accuracy of the approximation and the time it takes to simulate a model.

2.2 Model-Based Cyber-Physical Systems Engineering

Chapter 1 introduces some of the major challenges in the development of CPSs, and some of the key efforts in overcoming those challenges using model-based approaches. The selection of an appropriate modelling formalism for the representation of CPSs is particularly challenged by their inherent complexity and heterogeneity (Derler et al. 2012, Broy 2013). A CPS model must include representation of both software and physical processes, alongside computation platforms and networks. The feedback loop between computational elements and the physical world with which they interact encompasses sensors and actuators, physical dynamics and computation, software scheduling, and networks with contention and transmission delays. Modelling CPSs with reasonable fidelity requires the cooperation of a range of engineering disciplines and the production of models comprising many heterogeneous components.

Whilst rich DE models are appropriate for the representation of software controlling such systems, they are not well suited to the abstraction of the physical world with which the software interacts. Conversely, engineers of physical disciplines typically utilise CT techniques to represent physical processes but these formalisms are limited in their ability to express the complexities of control software. The lack of a common language for reasoning about properties of systems composed of heterogenous components often leads to misunderstandings and hinders the development of CPSs (Heemels & Muller 2007).

The development of CPSs is dependent on the cooperation of engineers from a range of complementary disciplines, each with its own methods, vocabularies, and styles of reporting (Verhoef et al. 2014). Each discipline has matured independently, and whilst model-based techniques are common to each, the types of model vary significantly. Model-based approaches to overcoming challenges associated with heterogeneity include model translation, the composition of modelling languages or models, and the combination of modelling tools (Hardebolle & Boulanger 2009). Techniques for the combination of modelling languages are often employed in the development of *hybrid systems*, described in Section 2.2.1. The combination of complementary models is embodied in *actor-oriented modelling*, described in Section 2.2.2. The combination of complementary modelling tools is described as *collaborative multi-modelling*, and this approach is described in Section 2.2.3.

2.2.1 Hybrid Systems Modelling

Continuous physical dynamics can be expressed using ordinary differential equations, and combined with descriptions of discrete behaviours expressed using finite automata. Such an integration directly combines complementary modelling formalisms, and is described as a *hybrid system* (Maler et al. 1991). In a hybrid system, each continuous state includes initial conditions for time, differential equations to describe the progression of the state, and invariants which describe areas of the continuous state-space where the system remains in a discrete state (Alur et al. 1994).

The definition of an effective hybrid systems modelling language is nontrivial. Such an approach must ensure that models of deterministic systems are themselves be deterministic, and not dependent on nondeterministic outputs of a differential equation solver. Also, the accurate representation of causally related but distinct events is particularly challenging when they occur simultaneously (Derler et al. 2012).

A variety of tools support the development and utilisation of hybrid systems models (Carloni et al. 2006). The provision of determinate semantics for hybrid systems models described by Lee & Zheng (2005), is realised in the open-source HyVisual tool (Brooks et al. 2005), and influenced the development of commercial tools including Simulink (Dabney & Harman 2004) and LabVIEW (Travis & Kring 2007).

Whilst a hybrid systems approach might be suitable for some simple CPSs, the ad hoc combination of continuous and discrete time in models limits the applicability of the approach for more complex CPSs.

2.2.2 Actor-Oriented Modelling

Model-based development of embedded systems is often approached by combining distinct computational models representing complementary system components (Lee et al. 2003). In such models, software components are typically described as *actors*, and actors execute concurrently and communicate by sending messages via interconnected ports (Lee 2003).

In combining diverse actor models, it is essential that all models share a common and well-defined semantics, including diverse models of time. Actor-oriented frameworks with a semantic notation of time such as Simulink (Dabney & Harman 2004) and Modelica (Fritzson & Engelson 1998), assume that time advances uniformly across an entire system, which limits their use in CPS modelling where distribution of components challenges this assumption. Other frameworks include a heterogeneous notion of time, utilising models such as Programming Temporally Integrated Distributed Embedded Systems (PTIDES; Zhao et al. 2007), for executing model events on multiple timelines but maintaining determinism for event interaction.

A composition of models approach with support for timeline multiplicity is adopted by the Ptolemy project[1], which focusses on modelling, simulation, and design of concurrent, real-time embedded systems using a combination of actor-oriented computational models (Bae et al. 2009, Eker et al. 2003). The Ptolemy approach facilitates coherent coupling of heterogeneous models, which includes tools to support simulation of both DE and CT models. Despite this, model constructs are primitive, and lack object-orientation for DE descriptions or component libraries for CT expressions.

2.2.3 Collaborative Multi-Modelling

The integration of distinct modelling tools is described as *multi-modelling* (Fishwick & Zeigler 1992, Mosterman & Vangheluwe 2004). Tool integration typically involves enabling interoperability of tools from independent vendors, which is often challenging and can result in fragile tool chains (Karsai et al. 2005). By combining established modelling tools, engineers are afforded the expressiveness of formalisms tailored for both DE and CT expressions separately. By the integration of diverse models

[1]See `https://ptolemy.berkeley.edu/`

in a single collaborative model (*co-model*), and facilitating its simulation (*co-simulation*), the system as a whole can be validated without compromising the expressiveness of either DE or CT descriptions.

Support for the development of co-models and the execution of co-simulations include an approach by Myers et al. (2011), who describe an approach to co-modelling and co-simulation specifically for the analysis of hardware/software partitioning, and Zhang et al. (2014), who propose a co-simulation framework aimed at the automotive domain.

An architecture for the design of more generic co-simulation tools is proposed by Nicolescu et al. (2007), which describes the linking of multiple simulators using a simulation bus. A variation of this architecture is used by the Crescendo tool (Larsen, Gamble, Pierce, Ribeiro & Lausdahl 2014), an output of the Design Support and Tooling for Embedded Control Software (DESTECS) project[2]. The DESTECS project coordinated the development of methods and tools for the creation and co-simulation of heterogeneous co-models. The DE and CT models integrated into a Crescendo co-model are designed and developed using their native formalisms and supporting tools, and combined using the Crescendo co-simulation tool. Crescendo co-models facilitate multidisciplinary modelling of CPSs and are unique in their coordination of distinct tools to facilitate co-simulation (Fitzgerald et al. 2014*b*).

Crescendo Co-Modelling and Co-Simulation

A Crescendo co-model is a composition of a DE model of a *controller* and a CT model of a *plant* (the parts of the system which are controlled), coupled with a co-simulation *contract* which specifies an interface for the exchange of information between the two models. Information is exchanged using *shared variables*. The value of a shared variable can be accessed from both constituent models, although can only be updated by one of them. Variables updated by the DE constituent model care described as *controlled* variables, whilst those updated by the CT constituent model are described as *monitored* variables (Fitzgerald & Pierce 2014). The basic components of a Crescendo co-model and their interactions are illustrated in Figure 2.1.

Shared variables are declared in the contract, which ensures that the constituent models are *consistant*. Model consistency comprises *syntactic consistency* and *semantic consistency*. Ensuring syntactic consistency includes explicit declaration of shared variables and their data types, whilst semantic consistency is aided by the provision of SI units or basic descriptions of the nature of the data described

[2]See `http://www.destecs.org/`

by each variable. Model consistency is essential for the production of trustworthy co-simulation results (Fitzgerald & Pierce 2014).

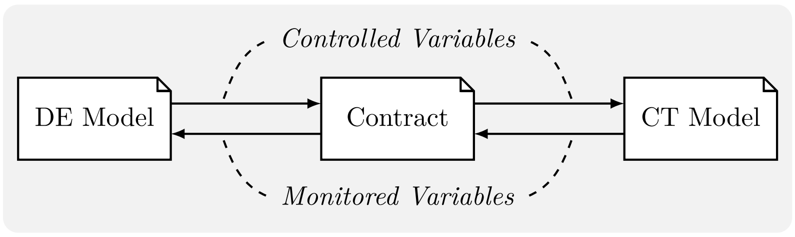


Figure 2.1: Crescendo Co-model Architecture

The DE constituent model of a Crescendo co-model provides a description of digital control, typically implemented in software. The design of software requires notations and tools with features for the description of system behaviours which result from a discrete series of actions and facilities to express the structure and logic of hierarchical architectures. The selected formalism for constructing DE models in the Crescendo framework is VDM (Larsen, Fitzgerald, Verhoef & Pierce 2014). VDM is a rich, general-purpose language for modelling discrete systems, and has three complementary dialects: The VDM Specification Language (VDM-SL), is designed for aiding the definition of functional specifications of systems. Data is defined in terms of basic data types, and functionality is described in terms of functions and operations over the data types (Fitzgerald et al. 2005). An object oriented extension (VDM++) augments VDM-SL to enable the specification of object oriented systems (Mukherjee et al. 2000, Verhoef et al. 2006, Verhoef & Larsen 2007), and a further extension of that (VDM-RT) introduces constructs for analysing real-time, embedded, and distributed systems (Verhoef 2009, Larsen et al. 2013). VDM modelling is supported by the *Overture* tool (Larsen et al. 2010), which can be used to construct, animate and analyse VDM models.

The constituent CT model of a Crescendo co-model provides a description of physical phenomena which are measured or influenced by digital control. The representation of physical systems requires notations and tools which can express phenomena typical of a variety of domains, including mechanical, electrical and hydraulic systems. The selected formalism for constructing CT models in the Crescendo framework is 20-sim (van Amerongen et al. 2014). 20-sim supports the representation of multidisciplinary physical systems using differential equations coupled with graphical iconic diagrams (Kleijn 2009) and domain independent bond graphs (Karnopp & Rosenberg 1968). 20-sim modelling is supported by the 20-sim tool, which facilitates visualisation and simulation of 20-sim models.

The Overture and 20-sim tools used to construct Crescendo co-models each includes a simulator which is used by the Crescendo tool to facilitate co-simulation. The independent DE and CT simulators have responsibility for the simulation of their respective constituent models, and the coordination of the simulators is the responsibility of a *co-simulation engine*. The Crescendo tool provides a co-simulation engine which is responsible for orchestrating the progression of time between Overture and 20-sim simulators and the propagation of data between the constituent models (Fitzgerald & Pierce 2014). The co-simulation engine takes the co-simulation contract as an input, and produces a set of results. The results describe the state of each model and the value of shared variables at synchronisation steps throughout the co-simulation, and 20-sim facilitates the plotting of these values on user-defined graphs.

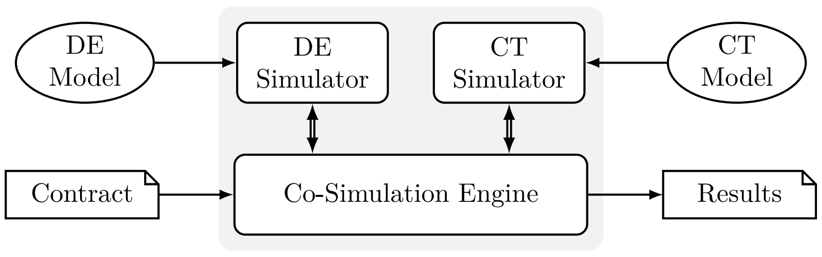


Figure 2.2: Crescendo Co-simulation Architecture

Interaction between elements in a co-simulation is illustrated in Figure 2.2. Independent DE and CT simulators take a DE and CT model respectively as input and communicate with the co-simulation engine, but do not interact directly. Double arrows between the DE/CT simulators and the co-simulation engine indicate the exchange of both synchronisation data and values assigned to shared variables.

Each simulator maintains its own local state and internal simulation time throughout a co-simulation, and the co-simulation engine facilitates the synchronisation of time between the simulators. Calculation is performed synchronously and at equal time steps by the two simulators, described as *lock-step co-simulation*. This is in contrast to *optimistic co-simulation*, a synchronisation scheme which sees each simulator progress at its own pace but requires resource-intensive rollback when one simulator progresses beyond the occurrence of an event in another simulator which requires the simulations to be synchronised (Broenink et al. 2010).

At the beginning of a co-simulation step the two simulators have a common simulation time. The DE simulator first updates the value of any controlled variables and calculates the time of the next state change in the DE model to determine the duration of the time step. The controlled variables and the proposed step duration are communicated to the CT simulator via the co-simulation engine. The CT simulator then attempts to advance to the time proposed by the DE simulator, but stops if an event occurs during the period. The actual time reached by the CT simulator is communicated back to the DE simulator along with the value of monitored variables via the co-simulation engine. The DE simulation then advances so that both simulators are again synchronised and the time step is complete (Coleman et al. 2014).

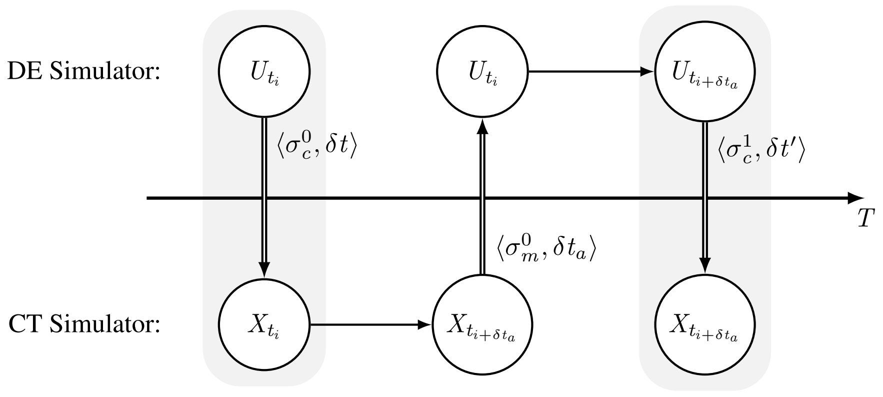


Figure 2.3: Lock-Step Co-simulation Synchronisation

This synchronisation scheme is illustrated in Figure 2.3. The internal state of the DE simulation at time t_{de} is described by $U_{t_{de}}$ and the internal state of the CT simulation at time t_{ct} described by $X_{t_{ct}}$. Vertical double arrows indicate the exchange of shared variables and proposed time steps. Though the co-simulation engine is not included in the diagram, it provides the mechanism for facilitating these exchanges.

The state of shared variables is encapsulated in σ, with the value of controlled variables defined in σ_c, and the value of monitored variables defined in σ_m. A time step is proposed of duration δt. Horizontal arrows highlight a state transition, where a simulator progresses its internal time.

At the beginning of the co-simulation synchronisation cycle, both simulators are synchronised to an initial time t_i. The DE simulator sets the controlled variables (σ_c^0), and a proposes a duration δt, by which the CT simulator should advance if possible. This data is communicated to the CT simulator via the co-simulation engine, and the CT simulator attempts to progress to time $t_i + \delta t$, but stops if an event is reached. The actual time reached by the CT simulator (δt_a) is passed back to the DE model via the co-simulation engine, along with the values of monitored variables (σ_m^0). If no events occur during this interval, then $\delta t_a = \delta t$, otherwise $\delta t_a < \delta t$. The DE simulator (still in its initial state) then progresses by δt_a, so that the two simulators are again synchronised, and the cycle repeats.

2.2.4 Observations

The discontinuous behaviour of software systems demands model-based descriptions which enable reasoning about large and complex state spaces. Placing reliance on such software requires confidence beyond that of testing selected sample cases, and DE models might employ a range of techniques to manage the complexity of software. Structuring techniques are essential for the organisation of models, and a degree of rigour is essential for enabling semantic analysis of models. Whilst many programming languages include mechanisms for structuring and have some degree of rigour, DE modelling languages also facilitate abstraction and enable early analysis of software prior to its implementation.

The continuous behaviour of physical systems demands model-based descriptions which enable representation of phenomena from contrasting domains, including mechanical, electrical and hydraulic systems. Models underpinned by differential equations can be analysed statically, but the impact of interactions between components is typically explored using simulation-based analyses.

For modelling systems which couple both DE and CT phenomena, a variety of approaches have been developed for overcoming the challenges of heterogeneity. The combination of DE and CT expressions in a single model is limited to the expression of hybrid systems which lack the complexity of DE/CT interaction common to CPSs. The combination of DE and CT models provides more comprehensive support for CPS modelling, but existing solutions are limited to the provision of primitive constructs for modelling both DE and CT phenomena. By combining modelling tools, co-modelling supports independent modelling of DE and CT phenomena using well established formalisms. By enabling engineers to model DE and CT components in diverse environments, both cyber and physical components can be represented without compromise. By co-simulation of co-models, behaviours which emerge through interaction of DE and CT elements can be analysed.

The co-modelling and co-simulation framework supported by the Crescendo tool enables models of cyber and phenomena to be expressed using DE and CT formalisms respectively, and the system analysed as a whole. DE models are expressed using the VDM formalism. VDM is particularly advantageous for the representation of DE systems, enabling abstract models which are structured hierarchically. It has a formally defined semantics, and enables systematic and automated analyses. CT models are expressed using the 20-sim formalism. 20-sim is well suited to CT systems modelling by its provision of domain independent notation, support for combining equation and iconic representations, and rich component libraries. By the provision of a co-simulation engine, behaviours which emerge as a result of the close interaction across the cyber/physical boundary can be analysed using co-simulation. Crescendo co-models combine VDM and 20-sims, both well-established formalisms with stable tool support and a record of industry use.

Whilst co-simulation technology is unique in its support for exploration of the interaction of rich models of software and continuous models of physics, it is limited in its application. So far, the approach has been restricted to enabling verification by co-simulation. More rigorous verification techniques are afforded by alternative approaches, alas this comes at the price of diminished expressiveness in models of DE and/or CT phenomena. Furthermore, whilst Crescendo co-models are domain independent, their origins in embedded systems modelling introduces an assumption that modelled systems are isolated entities. This is reflected in the coupling of a single DE model and a single CT model, with no consideration of how a system might interact outside of the system boundary. Whilst this is problematic for modelling cyber-physical constituents contributing to a SoS, VDM models support primitive virtual execution environments and communication mechanisms which can be harnessed for modelling interaction between distinct entities.

2.3 Model-Based Systems of Systems Engineering

Chapter 1 introduces some of the major challenges in the development of SoSs, and some of the key efforts in overcoming those challenges using model-based approaches. Engineering SoSs is particularly hindered by their unique combination of characteristics. Operational and managerial independence leads to limited knowledge and control of constituent systems, and distribution of constituents adds complexity to representing communication between them. Gaining confidence in the impact of emergent behaviours requires an approach which facilitates analysis of the system as a whole. Overcoming the unique challenges of SoS engineering can be aided by the use of model-based approaches (Maier 2005, Valerdi et al. 2008). Models of SoS architecture, constituent systems, infrastructure, and environment enable early exploration of design alternatives and the relationships between constituents, and facilitate validation of global properties of the SoS on which reliance might be placed.

A diverse range of approaches and formalisms have been employed in the model-based development of various aspects of SoSs (Nielsen et al. 2015), such as agent-based modelling for representing negotiation between consitituents (Acheson et al. 2013) and service-oriented modelling for managing system evolution (Zeigler & Zhang 2015). A variety of formal modelling languages have been used for the representation of SoSs (Caffall & Michael 2005). Languages underpinned by formal semantics enable machine-assisted analysis of global system properties. Examples of the application of formal languages for the representation of SoSs include the use of Event-B by Bryans et al. (2011). Event-B enables automated verification techniques and manages the complexity of proof by modelling SoSs as a chain of machines linked by refinement relations. However, this approach does not address interaction between participants, and does not provide an accessible representation of the SoS architecture.

Fitzgerald et al. (2012) argue that no single formalism is sufficient for representing all of the demands of SoS modelling, and propose the use of a combination of interoperable modelling techniques for SoS modelling. By employing a selection of techniques, complementary notations are used for effective modelling of architectural, interaction, and functional aspects. Specifically, they utilise SysML for developing architectural descriptions, Communicating Sequential Processes (CSP; Hoare 1978) for representing concurrency and communication, and VDM for data and functionality. This approach is extended by the Comprehensive Modelling for Advanced Systems of Systems (COMPASS) project[3], aimed to develop tools and techniques to support a formally grounded model-based approach to developing SoSs. Outputs of the COMPASS project include the COMPASS Modelling Language (CML; Woodcock et al. 2012) which combines CSP and VDM, and a CML extension for SysML. By linking SysML and CML models, models can be easily understood at a graphical level, but rigorously defined to support formal analyses.

2.3.1 Architectural Frameworks

The COMPASS approach promotes the use of architectural frameworks for the development of SoS architectural models. Since SoSs include a broad spectrum of application domains, a single framework for SoS modelling is avoided, and instead a framework for the development of architectural frameworks is proposed. The COMPASS Architectural Framework Framework (CAFF; Perry & Holt 2014) is intended to guide the development of architectural frameworks for a particular SoS application.

The CAFF is based directly on the Framework for Architectural Frameworks (FAF; Holt & Perry 2008, ch. 11). The FAF is a simple architectural framework indented to aid the production of further architectural frameworks. Architectural frameworks built according to this meta-framework can be specialised for application in a particular domain, or for a particular class of system. Concepts important to the development of architectural frameworks and relationships between them are based on concepts defined in the ISO standard *ISO 42010 – Systems and software engineering architecture description* (ISO/IEC/IEEE 42010 2011).

Concepts important to the development of architectural frameworks and relationships between them

[3]See `http://www.compass-research.eu/`

are based on concepts defined in the ISO standard *ISO 42010 – Systems and software engineering architecture description* (ISO/IEC/IEEE 42010 2011).

The definition of an architectural framework is achieved by the specification of a number of *viewpoints*. Viewpoints provide a specification for the presentation of information, and define what can be produced when the framework is used for the definition of an architecture. The realisation of a viewpoint for the definition of a particular architecture is described as a *view*. A view is an artefact which forms part of an architecture description, and describes a particular part of the architecture. An architecture which is based on an architectural framework comprises a series of views where each view conforms to a viewpoint in the architectural framework. Since the FAF is a meta-framework, the realisation of its viewpoints produces artefacts which are simultaneously *views* of the resulting architectural framework, and *viewpoints* for the definition of specific architectures according to the framework.

When used to support the production of an architectural framework, the FAF is intended to ensure that the requirements of the framework are properly understood and that relevant terminology is defined, necessary views of a system are understood, and any rules constraining the framework are identified. The FAF comprises six viewpoints to support these aims. To implement the FAF and create an architectural framework, these viewpints must be realised as views, which in turn provide the viewpoints of the resulting framework.

Architectural Framework Context Viewpoint (AFCV)

The Architectural Framework Context Viewpoint (AFCV) defines the context for the architectural framework. It represents and any concerns in context, to establish why the framework is needed.

The AFCV is intended to capture the needs and concerns of the architectural framework being defined. Realising this viewpoint requires identification of the needs and concerns which the architectural framework is designed to address, and any relationships between them. The roles of stakeholders involved in the definition of the architectural framework, or are relevant to its needs, should also be identified. Needs and concerns of the framework are encapsulated as *use cases* for the framework, and the outcomes of each use case should be observed by one or more stakeholders.

Production of the AFCV should be the starting point of the definition of an architectural framework, and it is central to the FAF. Realisation of the AFCV might be implemented using a SysML use case diagram, as demonstrated in Figure 2.4, a use case diagram which defines an example architectural framework context view for the FAF.

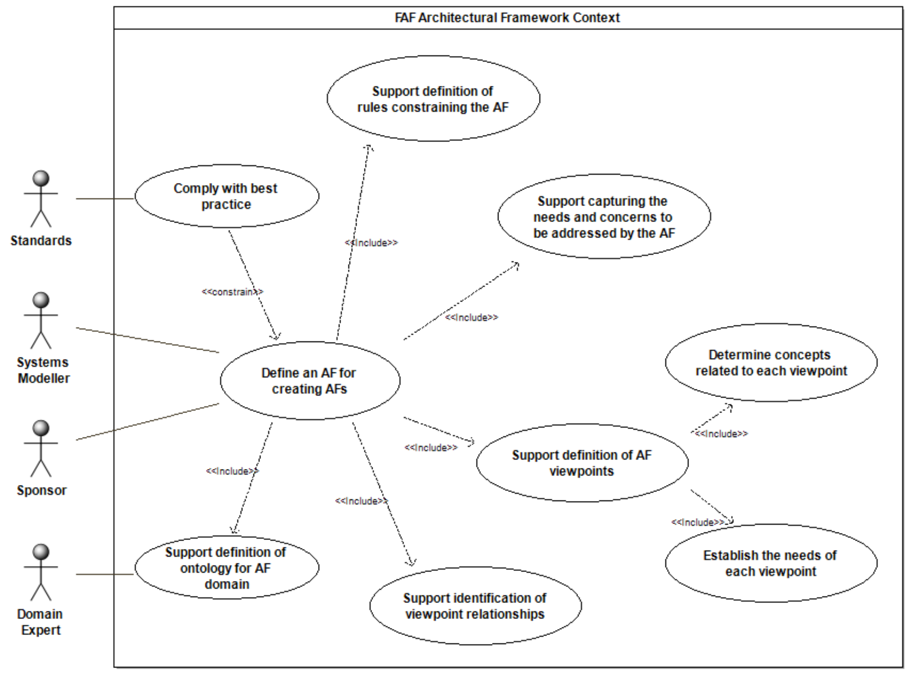

FAF Architectural Framework Context
Standards
Systems Modeller
Sponsor
Domain Expert
Comply with best practice
Support definition of rules constraining the AF
Support capturing the needs and concerns to be addressed by the AF
Define an AF for creating AFs
Determine concepts related to each viewpoint
Support definition of AF viewpoints
Establish the needs of each viewpoint
Support definition of ontology for AF domain
Support identification of viewpoint relationships
<<constrain>>
<<include>>

Figure 2.4: Architectural Framework Context View for Framework for Architectural Frameworks

Ontology Definition Viewpoint

The Ontology Definition Viewpoint (ODV) defines the concepts which might be included in a viewpoint and any relationships between concepts.

The ODV is derived from the information described by the AFCV, and is intended to support the definition of an ontology for the domain of the architectural framework being defined. Realising this viewpoint requires identification of important concepts of the domain in which the framework will be used, and any relationships between them.

Production of the ODV ensures that the definition of subsequent views uses a consistent set of related concepts, and might be implemented using a SysML block definition diagram, as demonstrated in Figure 2.5, a block definition diagram which defines an example ontology definition view for the FAF.

Viewpoint Definition Viewpoint

The Viewpoint Definition Viewpoint (VDV) defines each viewpoint using elements from the ODV.

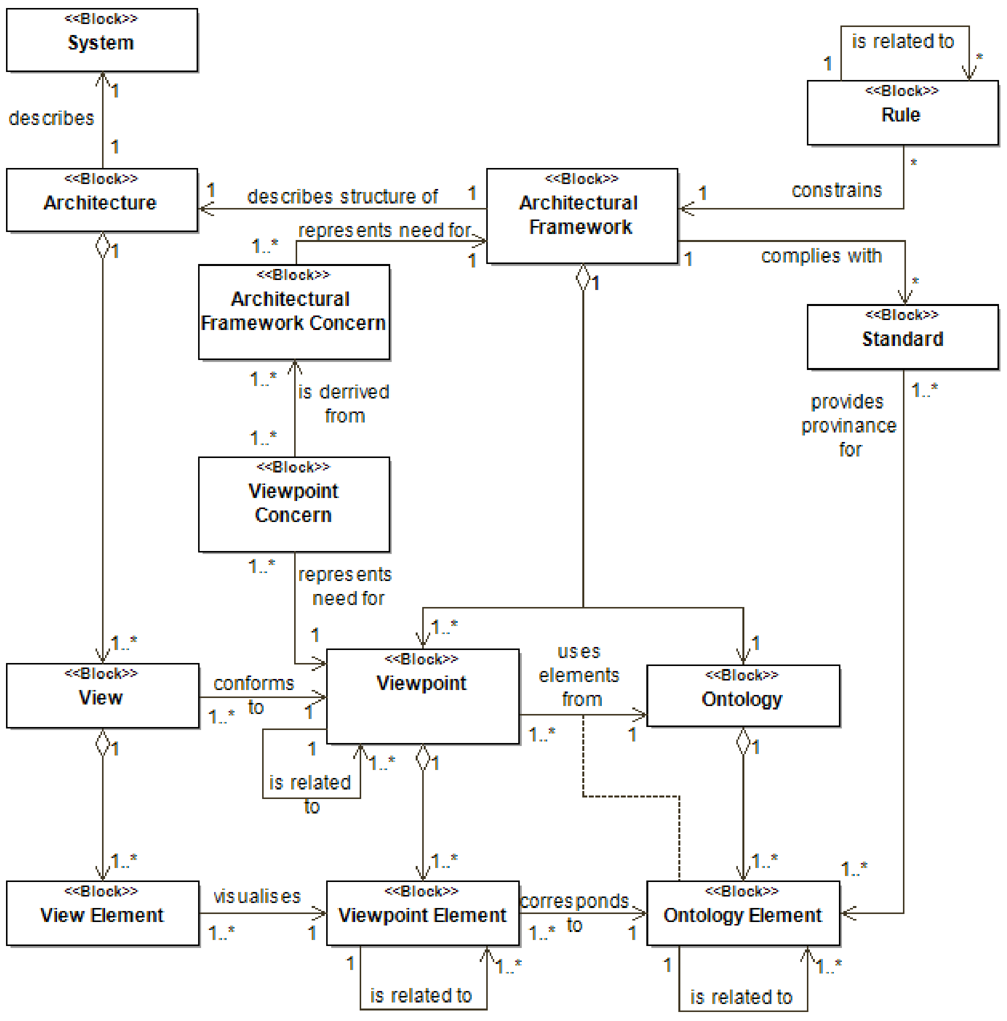


Figure 2.5: Ontology Definition View for Framework for Architectural Frameworks

The VDV is intended to support the definition of each viewpoint, and realising this viewpoint requires identification of elements and relationships from the ODV which are relevant to the viewpoint being defined. The definition of a viewpoint must be consistent with the particular needs of the viewpoint, which are defined in a Viewpoint Context View.

Realisation of the VDV might be implemented using a SysML block definition diagram, as demonstrated in 2.6, a block definition diagram which defines an exmaple viewpoint definition view for the FAF VDV.

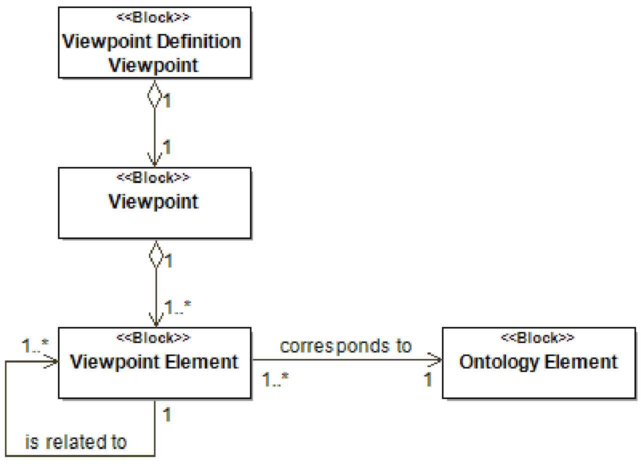


Figure 2.6: Viewpoint Definition View for Viewpoint Definition Viewpoint

Viewpoint Context Viewpoint

The Viewpoint Context Viewpoint (VCV) defines the context for each defined viewpoint using elements from the AFCV.

The VCV is intended to support the definition of the needs of each viewpoint, establishing the purpose of the viewpoint. Realising this viewpoint requires identification of the needs and concerns which the particular viewpoint is intended to address, and any relationships between them. The roles of stakeholders involved in the definition of the viewpoint or are relevant to its needs should also be identified. Needs and concerns of the framework are recorded as use cases that the outcomes of each use case should be observed by one or more stakeholders.

Realisation of the VCV might be implemented using a SysML use case diagram, as demonstrated in 2.7, a use case diagram which defines an exmaple viewpoint context view for the FAF VCV.

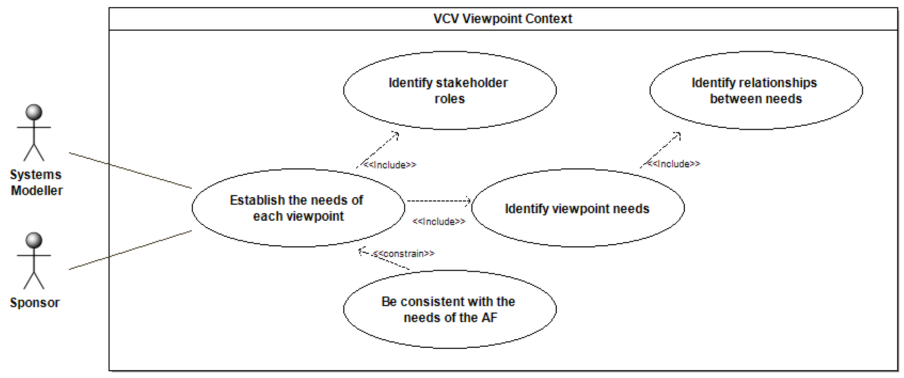

VCV Viewpoint Context
Identify stakeholder roles
Identify relationships between needs
<<include>>
<<include>>
Systems Modeller
Establish the needs of each viewpoint
<<include>>
Identify viewpoint needs
<<constrain>>
Sponsor
Be consistent with the needs of the AF

Figure 2.7: Viewpoint Context View for Viewpoint Context Viewpoint

Viewpoint Relationships Viewpoint

The Viewpoint Relationships Viewpoint (VRV) defines the relationships between the viewpoints which make up the architectural framework.

The VRV is intended to support the identification of relationships between viewpoints in the architectural framework being defined, and must be consistent with the needs of the architectural framework and its ontology.

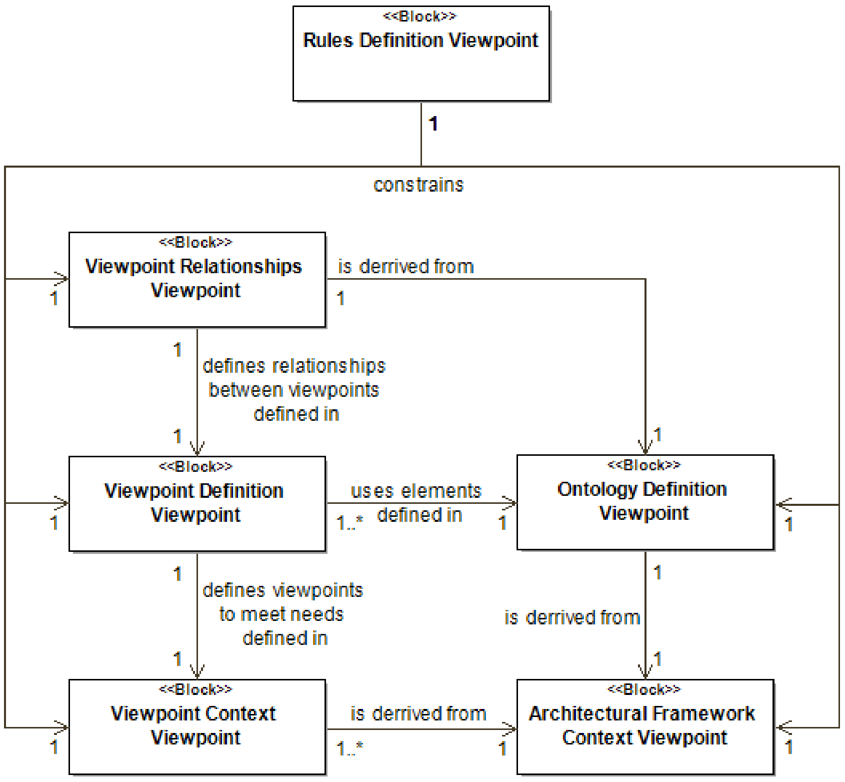
Realisation of the VRV might be implemented using a SysML block definition diagram, as demonstrated in Figure 2.8, a block definition diagram which defines an example viewpoint relationship view for the FAF.

Rule Definition Viewpoint

The Rule Definition Viewpoint (RDV) defines any rules which might constrain the architectural framework.

The RDV is intended to support the definition of rules to constrain the use of the architectural framework being developed, and realising this viewpoint requires identification of constraints of the framework and any relationships between them.

Realisation of the RDV might be implemented using a SysML block definition diagram, or more simply as a list of natural language contstraints, as demonstrated in Table 2.1, an example rule definition view for the FAF.



<<Block>>
Rules Definition Viewpoint
1
constrains
<<Block>>
Viewpoint Relationships Viewpoint
is derrived from
1
1
1
defines relationships between viewpoints defined in
1
<<Block>>
Viewpoint Definition Viewpoint
uses elements defined in
1..*
1
<<Block>>
Ontology Definition Viewpoint
1
1
1
defines viewpoints to meet needs defined in
1
1
is derrived from
1
<<Block>>
Viewpoint Context Viewpoint
is derrived from
1..*
1
<<Block>>
Architectural Framework Context Viewpoint
1
1

Figure 2.8: Viewpoint Relationships View for Framework for Architectural Frameworks

Table 2.1: Rules Definition View for Framework for Architectural Frameworks

FAF Rule 01
The definition of any architectural framework must implement at least one view for each of the six viewpoints.
FAF Rule 02
Each viewpoint in the Architectural Framework must be defined according to a Viewpoint Definition Viewpoint.
FAF Rule 03
Each Viewpoint Definition Viewpoint must be based on a corresponding Viewpoint Context Viewpoint.
FAF Rule 04
Each viewpoint in the Architectural Framework must be included in a Viewpoint Relationships View.

2.3.2 Observations

A multi-paradigm approach to SoS modelling is advantageous over the use of a single formalism, since the combination of techniques enables detailed investigation of disparate concerns. The utility of model-based approaches for the engineering of SoSs has been demonstrated for the description of architectures, testing and verification, and simulation.

Often concerned with the identification and exploration of interfaces, existing applications of model-based techniques to engineer SoSs are limited to the expression of interactions between digital systems.

The COMPASS approach combines complementary formalisms for the investigation of SoS architectures, functionality, and communication. Whilst the formalisms included in the approach are exclusive to the expression of discrete event systems, the provision of a meta-framework (CAFF) facilitates the extension of SoS architectural patterns to include mechanisms for the detailed representation of both cyber and physical constituents.

2.4 Summary

This chapter provides an overview of model-based techniques which might be employed in overcoming engineering challenges, including a detailed review of techniques employed for the representation of CPSs and SoSs.

Co-modelling is identified as an ideal candidate technology by its unique provision of executable models which enable the separate expression of DE and CT components using complementary formalisms. By the facilitation of co-simulation, co-models enable the exploration of emergent behaviours which result from interaction across the DE/CT boundary. The Crescendo co-modelling framework combines well established formalisms (VDM and 20-sim) for the development of models, and includes a co-simulation engine for their coordinated execution. Emerging from an embedded systems domain, existing techniques for the development of co-models are limited in their applicability to the domain of CPSoSs, by their lack of consideration for interaction between independent systems.

The organisation of independent systems demands a systematic approach to the development of system architectures. The use of architectural frameworks ensures such an approach to the description of architectures, but existing frameworks lack mechanisms for the detailed description of both cyber and physical systems. The development of an architectural framework is supported by the CAFF, a meta-framework for the development of bespoke frameworks.

Research Methodology 3

In order to achieve the research aims outlined in Chapter 1, appropriate research methods must be employed. A research methodology describes an approach to systematically solve a research problem and is used to motivate the selection of the most appropriate research methods (Kothari 2004).

A methodology serves as an intermediary between philosophy and scientific methods, and is intrinsically tied to the philosophical perspective of the research (Brown 2009). Whilst methods describe potential solutions to solving a particular problem, philosophy outlines the characteristics of the problem to be solved. A methodology describes a set of *principles of method*, which is used to inform the selection of methods which are uniquely suitable in solving a given problem, based on an underlying philosophical paradigm (Checkland 1981, p. 162).

This chapter presents a consideration of appropriate philosophical paradigms in terms of their ontological and epistemological underpinnings. By selection of an appropriate philosophical framework, methods for undertaking the research can be properly derived. Section 3.1 describes relevant philosophical principles before aligning the research described in this thesis with a philosophical framework. Section 3.2 introduces a range of methods for undertaking research, and considers a subset of methods which are most appropriate within the chosen philosophical framework.

3.1 Research Philosophy

Philosophy describes a system of beliefs which originate from the study of the fundamental nature of knowledge, reality, and existence (Waite & Hawker 2009, p. 685). Research should be conducted under guidance of a philosophical framework which outlines assumptions about the world and our perceived knowledge of it. A philosophical standpoint is based upon underlying ontological and epistemological assumptions. A philosophical ontology outlines the nature of reality, and comprises a set of beliefs about what is being studied. An underlying epistemology concerns the nature of knowledge, and describes what we can know about the reality described in an ontology (Lee & Lings 2008).

A spectrum of philosophical standpoints presents opposing ideas about the fundamental nature of knowledge and reality (Morgan & Smircich 1980, p. 492). At the extremes of the philosophical spectrum are *positivism* and *interpretivism*, the former with roots in natural sciences and the latter owing to the emergence of social science (Smith 1983).

Positivism is based on *philosophical realism*, and the assumption that reality is singular and objective. It is underpinned by the belief that reality is independent of a researcher, and remains unaffected by any investigation. A positivist researcher derives knowledge from objective evidence about observable and measurable phenomena, where rationally justifiable assertions can be supported by logical or mathematical proof (Walliman 2011). Whilst positivism enables the development of theories as a basis of understanding the world, its underlying assumptions are often considered inappropriate for investigation of social phenomena. The limitations of a positivist outlook led to the adoption of interpretivist approaches in the late 19th century (Smith 1983).

Interpretivism is based on *philosophical idealism*, and the assumption that reality is multiple and subjective, and is socially constructed. It is underpinned by the belief that reality is shaped by perceptions, and cannot be investigated without being influenced. An interpretivist researcher derives knowledge from subjective evidence, and concerns findings which are obtained from interpretation of qualitative data (Corbin & Strauss 2008).

Positivism and interpretivism represent extremes of a spectrum of research paradigms. Many additional paradigms can be placed between these extremes, each distinguished by the philosophical assumptions by which they are underpinned. Morgan & Smircich (1980) describe how a continuum of paradigms can exist simultaneously, and moving along the continuum gradually displaces the characteristics and assumptions of one paradigm with those of the next.

Contrary to suggestion that research should be undertaken within the bounds of a single paradigm, many argue that an effective researcher should be free to work under the assumptions of whichever paradigm is most appropriate for a particular study (McKerchar 2008). *Pragmatism* describes a philosophical framework which enables a researcher to combine philosophical considerations from complementary paradigms, selecting methods solely on the basis of their utility in addressing a given research question. Such an approach offsets the weaknesses of one paradigm with the strengths of another, and allows research to cross the boundary between positivism and interpretivism (Curran & Blackburn 2001).

Critics of pragmatism claim that by failing to commit to a single paradigm, a methodology lacks the support of a theoretical framework. Such critics contest that positivist and interpretivist paradigms operate under incompatible ontological and epistemological assumptions (Bryman 1984). However, Dzurec & Abraham (1993) suggest that inherent differences between approaches are outweighed by their similarities, and that differences in approach are not a result of conflicting objectives, rather that of contrasting strategies of complementary researchers. Onwuegbuzie & Leech (2005) argue that discrepancies in approach are best overcome by methodological pluralism facilitated by a pragmatist paradigm, and that relying on a single research paradigm introduces limitations which hinder the advancement of science.

Understanding philosophical principles is important throughout the research process. For example, on conducting a thorough literature review, one is likely to consider studies with contrasting philosophical standpoints. The evaluation of such diverse studies requires a basic understanding of the principles on which each work is based to avoid dismissing or misusing it (Lee & Lings 2008). Furthermore, an understanding of alternative philosophical positions not only enables critical evaluation of the impact of any assumptions made throughout the research process, but also helps to circumvent inconsistencies between a given methodology and any underlying assumptions.

3.1.1 Research Philosophy and Systems Engineering

As systems engineering is challenged by increasingly complex systems, inclusion of human and social elements has moved systems engineering research from the domain of natural sciences to include aspects fundamental to social sciences. Because of this shift, it is particularly important that systems engineering research methodology is based on a sound understanding of philosophical principles.

Brown (2009) presents a summary of approaches to methodology in systems engineering, and claims that the reputation of the discipline is commonly undermined by a lack of methodological rigour, attributed to confusion between methods and methodology. In a systematic review of papers submitted to the Conference on Systems Engineering, Brown (2009) identifies that it is commonplace for systems engineering researchers to simply describe their methods or techniques in place of a theoretically underpinned methodology.

Research drawing only superficially (if at all) from an established philosophical position is limited to employing a methodological approach which combines techniques through a process of trial and error, described by Midgley (1997) as *atheoretical pragmatism*. It is important for systems engineers to establish sound methodologies based on a philosophical standpoint; without doing so, research can only guide subsequent work on the basis on imitation rather than logical deduction.

3.1.2 Philosophical Positioning

Given the intersection of natural and social sciences within which systems engineering often falls, it is counterintuitive to constrain such research to a single philosophical paradigm. This suggests that such research is best suited to a pragmatist approach, rather than committing to a single system of philosophy.

For the selection of a well-informed methodology, it is important to consider any assumptions inherent to the research in question. In Chapter 1, it is claimed that it is possible to verify properties of CPSoS, and a number of intermediate objectives which are necessary in achieving this goal are presented. The research relies on the development of an approach which enables the combination of existing model-based engineering techniques.

The selection of appropriate techniques on which to build should be objective, and based on clear requirements. A systematic review of available modelling techniques is outlined in Chapter 2, where a positivist approach was necessary to assess the suitability of techniques based on the documented needs of CPSoS stakeholders. Similarly, a positivist approach is suited for the design of mechanisms for the facilitation of combining existing techniques. In contrast, the evaluation of the success of the proposed approach must facilitate the subjectivity of any claims; methodological approaches more commonly associated with interpretivism are likely to be more appropriate here.

For systems engineering research, it is not necessarily appropriate or possible to specify an overarching philosophical alignment with which each stage of the research should conform. Despite this, it is important that on undertaking the various stages, any ontological and epistemological assumptions which underpin the study are acknowledged.

3.2 Research Methods

A methodology serves to inform the selection of scientific methods based on an underlying philosophical paradigm. Methodological approaches embody a series of scientific methods, where methods describe the specific tools and techniques used for collecting and analysing data (Checkland 1981).

Positivist research typically employs methods of analysis based on the statistical analysis of quantitative data (Collis & Hussey 2013), and methodological approaches typically associated with positivism include:

Experimental studies are used to identify causal relationships by monitoring the effect that manipulation of an independent variable has on a dependant variable. Experimental studies are conducted in a systematic way in a laboratory or a natural environment, and are their design is dependent on the number and nature of data samples and the timing of experiments (Kervin 1995). The validity of findings from experimental studies is often undermined by the availability of appropriate samples, and in the case of laboratory experiments, the impact of an artificial setting.

Surveys are designed to collect data from a sample in order to generalise statistical findings to a larger population. Surveys might be *descriptive*—designed to provide an accurate representation of some phenomena—or *analytical*—designed to determine whether a relationship exists between variables. Statistical techniques are employed to predict the likeliness that characteristics of the sample are true of the full population.

Interpretivist research typically employs methods of analysis based on the interpretation of qualitative data (Collis & Hussey 2013), and methodological approaches typically associated with interpretivism include:

Case studies are empirical enquiries used to explore a single phenomenon in a natural setting using a range of methods to gain in-depth knowledge. Case studies can be used to describe an existing practice, to illustrate a new and innovative practice, to identify challenges in implementing new procedures and evaluate their benefits, or to understand and explain a scenario (Ryan et al. 2002).

Grounded theory describes a framework for the collection, coding and analysis of data using a systematic set of procedures to inductively derive a theory about phenomena. Grounded theory challenges positivist approaches which rely on *a priori* theories, instead generating theories based on generated data (Glaser & Strauss 1967).

The methodological approaches outlined here includes a small subset of examples from a vast catalogue of techniques. Whilst the approaches are described in the context of a particular philosophical paradigm with which they are typically associated, many of the included approaches are appropriate for use within a range of contrasting paradigms.

Data collected from a series of complementary research methods can be combined through a process of *triangulation*. Triangulation describes the combination of multiple sources of data, research methods, and/or researchers in an attempt to investigate a single phenomenon. Four categories of triangulation are identified by Easterby-Smith et al. (2012): *Theory triangulation* describes the use of some theory developed in one discipline and used in another; *Methodological triangulation* describes the use of a range of methods to collect and analyse data; *Data triangulation* describes the use of data collected from multiple sources or at different times; *Investigator triangulation* describes the use of multiple researchers independently collecting, sharing, and comparing data.

Methodological triangulation typically describes a combination of methods from a single research paradigm. Where a combination of methods from complementary paradigms is used, the approach is instead described as *mixed methods* (Collis & Hussey 2013). Mixed methods and triangulation can help reduce bias in any findings (Jick 1979), and when a range of methods lead to the same conclusions of a phenomenon, both the validity and reliability of the findings is increased (Denzin 1978).

3.2.1 Research Methods and Systems Engineering

The identification of appropriate methods is given little attention throughout engineering disciplines, and there is limited literature describing how engineers do research. Whilst considerable effort has been invested in many non-engineering disciplines to rigorously define appropriate research methods, engineering researchers tend to mimic methodological approaches from other fields (Ferris 2009).

Systems engineering research addresses a uniquely diverse range of challenges, covering a much broader set of concerns than those addressed in other engineering disciplines which are typically constrained by either a particular class of phenomena, technologies, or application area (Honour 1999). A systems engineering researcher is often challenged to consider technical, management, and product issues, where a product might be considered in terms of design, development, delivery, maintenance, and retirement of the system. Systems engineering research must also consider a highly diverse range of application areas, typically combining both technical and human considerations.

Since the subject matter of interest is much broader and multi-dimensional than in other engineering disciplines, Ferris et al. (2005) suggests that it is reasonable to expect that a wider range of research methods are appropriate for systems engineering research, and that such studies should not be limited to the range of methods usually employed in other engineering research.

By consideration of a taxonomy of philosophies, Ferris (2009) describes a process for the selection of methods based on the type of knowledge sought, and uses this as a basis for suggesting methods appropriate for systems engineering research. These methods include *positivist hypothesis testing*, *action research*, and *design*. In this context, Ferris (2009) describes these methods in the following ways:

Positivist hypothesis testing is an approach to research where some hypothesis is tested as an explanation of an observable phenomenon. This approach is typically employed in order to establish a contribution to theory which is valued as knowledge and is verifiable to some measure of statistical confidence. Knowledge gained from this approach is aligned with the paradigm of the particular field of study which enabled the original observation, the construction of the hypothesis, and the means of observation and the interpretation of the results.

Action research involves a researcher intervening in some activity and observing the effect of the intervention, in an attempt to identify an ideal intervention for ensuring some desirable outcome. This approach is typically employed in order to improve an established practice, and is heavily influenced by the impact of researcher and the research process on the scenario under investigation. Given this, knowledge gained from this approach is aligned with an interpretivist paradigm, and is specific to the particular situation in which it was developed.

Design describes a process where a researcher addresses an important and novel problem by designing a solution. This approach is typically employed in order to develop knowledge of practical application, but can also lead to the development of new or existing theories. Knowledge gained from design research is aligned with the paradigm of the engineering discipline, with novelty in the particular problem addressed and/or the combination of methods used in the proposed solution.

A diverse range of methods might be appropriate for undertaking systems engineering research. Section 3.2.2 proposes a structure for the investigation of the hypothesis defined in Chapter 1. The research is structured by the identification of a series of necessary tasks, where an appropriate method is identified for the completion of each task. The selection of methods is based on underlying philosophical considerations.

3.2.2 Research Design

Research design describes the planning of procedures for undertaking study in an attempt to maximise the validity of any findings (Vogt & Johnson 2011). A research design comprises a plan, structure and a strategy of investigation for obtaining answers to research questions, where the plan includes an outline of any tasks from writing hypotheses to the final analysis of the study (Kerlinger 1986). Further to the provision of a series of tasks, a research plan also ensures that the tasks are sufficient to obtain valid, objective, and accurate answers to any research questions (Kumar 2014). An effective research design should provide a framework for structuring the research process, and ensure consistency between chosen methods and a philosophical paradigm (McKerchar 2008).

Chapter 1 introduces a number of objectives which must be satisfied in order to address the given hypothesis. An overview of the proposed research identifies the requirements of a systematic approach to meeting these objectives. The proposed approach to model-based design of CPSoSs requires the specification of two complementary abstractions of a system of interest: an *architectural description* and an *executable co-model*.

An architectural description facilitates the identification of constituent systems and their characteristics, as well as any relationships between them. To support the production of an architectural description, the use of an architectural framework ensures that any important details are included. For CPSoS modelling, an architectural framework must facilitate the description of both cyber and physical constituents, their interfaces, and any interactions between them. Consideration of the suitability of existing architectural frameworks and identifies that a bespoke framework is required for the design of CPSoSs.

Whilst an architectural Framework provides a structured approach to the development of an architectural description, it does not specify how the information should be recorded. A wide range of systems modelling notations might be used for realising an architectural description, but for CPSoS modelling the chosen formalism must support the description of systems which are composed of a number of constituents, and each constituent might be expressed in terms of discrete or continuous phenomena. The chosen notation must also support the specification of system interfaces and any interactions between constituents. Consideration of the use of general purpose modelling languages identifies that SysML is the most appropriate language for expressing architectural descriptions of CPSoSs, with mechanisms for the specification of both structural and behavioural constructs of both discrete and continuous systems.

Further to an architectural description, the approach must support the development of an executable co-model of a system of interest. By facilitating simulation, executable co-models of CPSoSs enable the verification of properties which emerge as a result of interaction between constituents, and the integration of discrete event and continuous time phenomena. Consideration of heterogeneous co-modelling techniques identifies the use of Crescendo co-models as the most appropriate formalism for expressing the behaviours of CPSoSs.

It is important that the characteristics of a modelled CPSoS are consistent in both architectural and executable models. In order to ensure that the refinement from a SysML description to a co-model preserves the semantics of the original specification, the process should be in some way automated. An automated translation between notations should be based on well-defined translations. The VDM-SL specification language can be used to provide an abstract description of the underlying meta-models of both SysML and co-model representations. By encoding the information encapsulated by the meta-model of each language, a function can be defined which maps a SysML model to a co-model. By the definition of this mapping, the translation is demonstrated as appropriate for automated refinement within a suitable open tools framework.

To gain confidence in the proposed approach, it is important that it is both verified and validated. To verify the proposed techniques, they should be applied in the engineering of a suitable case study. Demonstration of the proposed approach is essential in enabling evaluation of its effectiveness. To assess the utility of the approach, its impact on baseline technologies should be assessed. The suitability of utilised technologies should be assessed using a suitable technology benchmarking framework.

Given the above consideration of the facets of a systematic approach, a series of tasks necessary for its specification are summarised below. For each task, an appropriate methodological approach is identified, based on any underlying philosophical considerations.

Development of a bespoke architectural framework This task requires a novel solution to an existing problem, where the scope of the problem is identified by experienced engineers. The requirements of the framework are derived from relevant literature (e.g. Hafner-Zimmermann & Henshaw (2017)), and are considered objectively. Given this, the design of the framework is typical of positivist research, and is suitable within the scope of the accepted requirements.

Development of a SysML Profile to support the architectural framework Similarly to the development of an architectural framework, this task requires a novel solution to a well-defined problem. The requirements of the profile are determined by the ontology specified in the architectural framework, and as such are objective, singular, and suited to a positivist approach.

Definition of a specification for mapping SysML descriptions to co-models Another design activity, the definition of a specification for mapping SysML models to co-models is twofold: firstly, an abstract representation of the information encapsulated by each model type must be defined; secondly, a function for mapping information contained within the SysML meta-model to the structure of a co-model meta-model must be provided. The mapping should be complete for all SysML models which are correct with respect to the given SysML profile, and is typical of research within a positivist paradigm.

Verification of the approach Verification of the approach must objectively evaluate the effectiveness of the proposed techniques by their application. A case study should be developed that exercises the approach in the development of a candidate CPSoS, and the success of the approach should be assessed by consideration of its ability to satisfy verification criteria. An interpretivist research approach is required in the evaluation of the effectiveness of the approach, to minimise the impact of subjectivity on the validity of the assessment.

Validation of the approach Validation of the approach must objectively evaluate the utility of the proposed techniques by consideration of the impact of the approach on the suitability of baseline technologies for the engineering of CPSoSs. Mechanisms for the evaluation of technology suitability and maturity should be employed in the evaluation of the approach, such as technology readiness assessments and technology benchmarking frameworks. An interpretivist approach is required in the assessment of the approach utility, to manage the impact of subjectivity on any evaluation.

The propsed methods for the undertaking of each of the tasks required for undertaking the proposed research is summarised in Table 3.1.

Table 3.1: Method selection per task

Task	**Methodological Approach**
Architectural framework development	Design
SysML profile development	Design
Translation specification	Design
Approach verification	Case study
Approach validation	Benchmarking

3.3 Summary

A research methodology describes an approach to systematically solve a research problem, and serves as an intermediary between philosophy and scientific methods. By consideration of underlying epistemological and ontological assumptions, a methodology is used to inform the selection of research methods based on an underlying philosophical paradigm. Effective research design provides a structure for undertaking appropriate research, and ensures consistency between chosen methods and a philosophical paradigm. This chapter presents a series of tasks necessary for investigating the hypothesis outlined in Chapter 1, and describes the selection of research methods for the completion of each task based on an underlying philosophical framework.

Section 3.1 presents a spectrum of philosophical paradigms and the implications of working within a particular paradigm. A consideration of philosophical alignment within the domain of systems engineering is described, before identification of the most appropriate paradigm with which to align the research described in this thesis. Section 3.2 introduces a selection of scientific methods which might be considered for the undertaking of the research, and assigns the most appropriate method for each of a series of tasks considered necessary for the successful completion of the work.

The undertaking of research tasks using the designated methods is described in subsequent chapters. The development of an architectural framework for CPSoS, and a SysML profile for the realisation of architectural descriptions are described in Chapter 4. Chapter 5 outlines the underlying meta-model for both the SysML profile and the co-modelling framework. This meta-model description is used in the specification of a translation between formalisms. Verification of the approach by its application to a case study is described in Chapter 6, and the utility of the approach is validated by the consideration of technology benchmarks is described in Chapter 7.

4 Describing Architecture

Architectural descriptions are an invaluable tool for the development of complex architectures, and are widely used in the engineering of SoSs. The development of architectural descriptions can be aided by the use of an architectural framework, which describes a number of viewpoints. Viewpoints encapsulate particular properties of a system, and complementary viewpoints contribute to a complete description of a system. By ensuring the inclusion of a minimum set of considerations in the definition of a system architecture, an architectural framework supports the production of architectural descriptions which are sufficiently detailed for use in a particular domain.

The realisation of viewpoints requires the use of a suitable architectural description language. An appropriate language can be specialised for the realisation of particular viewpoints by the provision of a language profile. A widely used and domain-independent language, SysML includes extensibility mechanisms to tailor its use for a particular application, such as the realisation of domain-specific architectures.

This chapter describes the development of an architectural framework for the development of architectural descriptions of CPSoSs in Section 4.1, and the definition of a supporting SysML profile in Section 4.2.

4.1 Architectural Framework Design

This section describes the development of an architectural framework to aid the development of architectural descriptions of CPSoSs. The framework is developed using the COMPASS Architectural Framework Framework (CAFF) described in Chapter 2. The CAFF is a meta-framework intended to aid the design of further architectural frameworks, and comprises a set of six viewpoints which should be realised in the definition of a framework:

Architectural Framework Context Viewpoint encapsulates the requirements of the framework, and its realisation is intended to ensure that the requirements of the framework are properly understood.

Ontology Definition Viewpoint outlines terminology integral to the framework, and defines concepts which can be modelled using the framework.

Viewpoint Relationships Viewpoint summarises the viewpoints of the framework and relationships between them.

Viewpoint Context Viewpoint identifies the requirements of a given viewpoint, and how they relate to the overall requirements of the framework.

Viewpoint Definition Viewpoint describes terminology and concepts from the framework ontology which is related to a particular viewpoint.

Rules Definition Viewpoint defines a set of rules constraining the framework.

The architectural framework for CPSoSs is defined by the realisation of the CAFF viewpoints. The definition of the framework requirements is supported by the realisation of the Architectural Framework Context View in Section 4.1.1, and domain-specific terms and concepts are identified by the realisation of the Ontology Definition View in Section 4.1.2. These requirements and concepts are used in the definition of framework viewpoints in Section 4.1.3, by the realisation of the Viewpoint Context View and Viewpoint Definition View. Additionally, the relationships between the specified viewpoints are defined by the realisation of the Viewpoint Relationships View. Finally, constraints over the framework are outlined by the realisation of the Rules Definition View in Section 4.1.4. The definition of the proposed framework by realisation of the CAFF viewpoints is summarised in Appendix A.

Throughout this chapter, the views of CPSoS-AF are typically described diagrammatically. Whilst it is not imperative that any particular notation is adopted for defining a framework, where appropriate the views of the proposed framework are expressed using SysML to be consistent with the definition of CAFF, where the authors also use SysML for defining framework views.

4.1.1 Establishing Requirements

The production of an architectural framework is intended to support the engineering of CPSoSs by ensuring a systematic approach to the development of architectural descriptions. A systematic approach to the description of architectures is important in ensuring that architectural models are sufficiently descriptive to support the engineering of more concrete representations of architectural elements.

The principal requirement of the framework to support the engineering of CPSoSs can be decomposed into further requirements by consideration of stakeholders likely to be involved in the definition of CPSoS architectures. The requirements of each stakeholder should be included in the requirements of the framework. In the engineering of CPSoSs, likely stakeholders include:

Systems Architect: The role involved with the organisation of systems composed of constituent systems. For the architectural framework to be employed, it is likely that a system has been identified as a CPSoS and a systems architect appointed. The success of this role requires support for the decomposition of a system to enable the identification of constituents, and the identification of any relationships between them.

Software Engineer: The role involved with the identification of constituent systems most appropriately expressed using a discrete event formalism. CPSoSs include digital systems, typically implemented in software, and the structure and behaviour of these systems should be included in an architectural description of the system. This role requires support for the definition of digital constituents, their interfaces, and their behaviour.

Domain Engineer: The role involved with the identification of constituent systems most appropriately expressed using a continuous time formalism. CPSoSs include systems comprising physical components from a particular engineering domain (e.g. electrical, mechanical), and the structure and behaviour of these systems should be included in an architectural description of the system. This role requires support for the definition of physical constituents, their interfaces, and their behaviour.

Standards and Guidelines: The role of any appropriate standards of guidelines for the development of a given system. Depending on the system being described, a range of processes or documents might outline constraints for the development of the system. These constraints should be considered in the development of an architectural description to ensure the developed system complies with best practice.

The identification of stakeholders of a CPSoS architectural description and their requirements of a supporting architectural framework realises the Architectural Framework Context View of the CAFF. This can be represented diagrammatically using a Use Case Diagram, as illustrated in Figure 4.1.

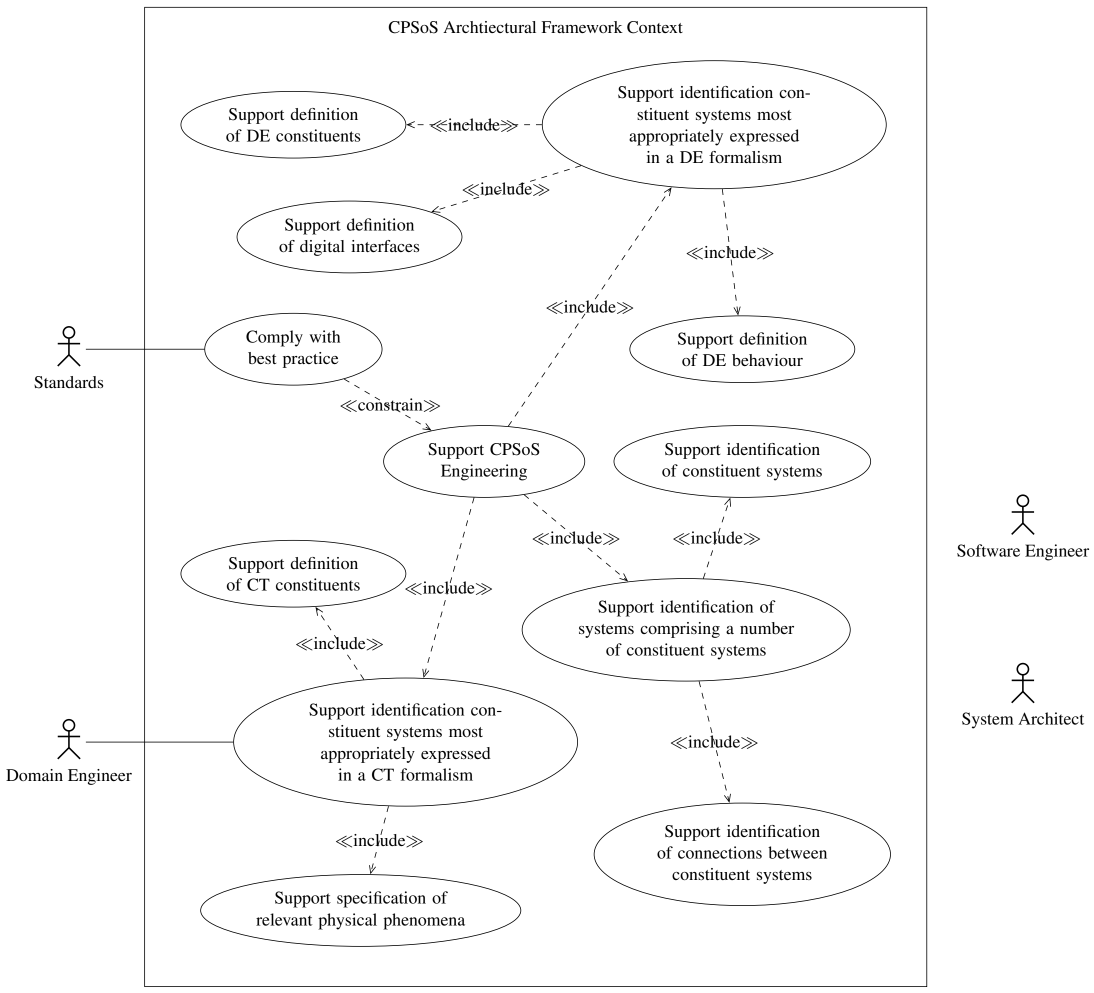

CPSoS Archtiectural Framework Context
Support definition of DE constituents
≪include≫
Support identification constituent systems most appropriately expressed in a DE formalism
≪include≫
Support definition of digital interfaces
≪include≫
≪include≫
Support definition of DE behaviour
Standards
Comply with best practice
≪constrain≫
Support CPSoS Engineering
Support identification of constituent systems
≪include≫
≪include≫
Software Engineer
Support definition of CT constituents
≪include≫
Support identification of systems comprising a number of constituent systems
≪include≫
System Architect
Domain Engineer
Support identification constituent systems most appropriately expressed in a CT formalism
≪include≫
≪include≫
Support identification of connections between constituent systems
Support specification of relevant physical phenomena

Figure 4.1: Architectural Framework Context View for CPSoS-AF

4.1.2 Framework Ontology

To support the development of architectural descriptions of CPSoSs, it is important to identify concepts important to their definition. The identification of these domain-specific concepts and the relationships between them constitutes an ontology for CPSoS engineering. The identification of concepts is an iterative process, where the design of viewpoints to satisfy particular stakeholder requirements typically demands the inclusion of specific concepts. Concepts important to the description of CPSoSs include:

System: Different types of *System* include *CPSoS*, *Composite*, and *Elementary*. Systems might interact with other systems via an *Interface*, typically exposed by a series of *Ports*.

CPSoS: A *CPSoS* is a type of system which is composed of a collection composite and/or elementary systems.

Composite System: Similarly to a CPSoS, a *Composite System* is composed of further constituent composite and/or elementary systems.

Elementary System: An *Elementary System* is not a composition of additional systems, but encapsulates some phenomena. An elementary system is either a *Cyber* or *Phyiscal* constituent.

Cyber System: A *Cyber System* is typically implemented in software, so comprises a series of *Classes*. A *Class* is a software entity, and is a template for the creation of software objects. Classes are composed of *Attributes* for the recording of data and *Operations* for the manipulation of attributes and/or invocation of additional operations. Classes can be specialised, such as the *Control Class*, which is responsible for the definition of the behaviour of given cyber system.

Physical System: A *Physical System* describes some continuous time phenomena using *Equations* acting over *Variables*. Equations calculate the the value of variables over time, and variables might also be updated by operations of cyber systems.

Interface: An *Interface* facilitates interaction between constituent systems. Cyber systems might interact with additional cyber systems via a *Service-based Interface*, and/or with physical systems via a *Cyber-Physical Interface*. A *Sense Interface* describes a cyber-physical interface for the transfer of sensed data from a physical system to a cyber system, whilst an *Actuate Interface* describes a cyber-physical interface for the transfer of control data from a cyber system to a physical system. Physical systems might interact with other physical systems via a *Flow-based Interface*.

The identification of terms and concepts important for CPSoS engineering realises the Ontology Definition View of the CAFF. This can be represented diagrammatically using a Block Definition Diagram, as illustrated in Figure 4.2.

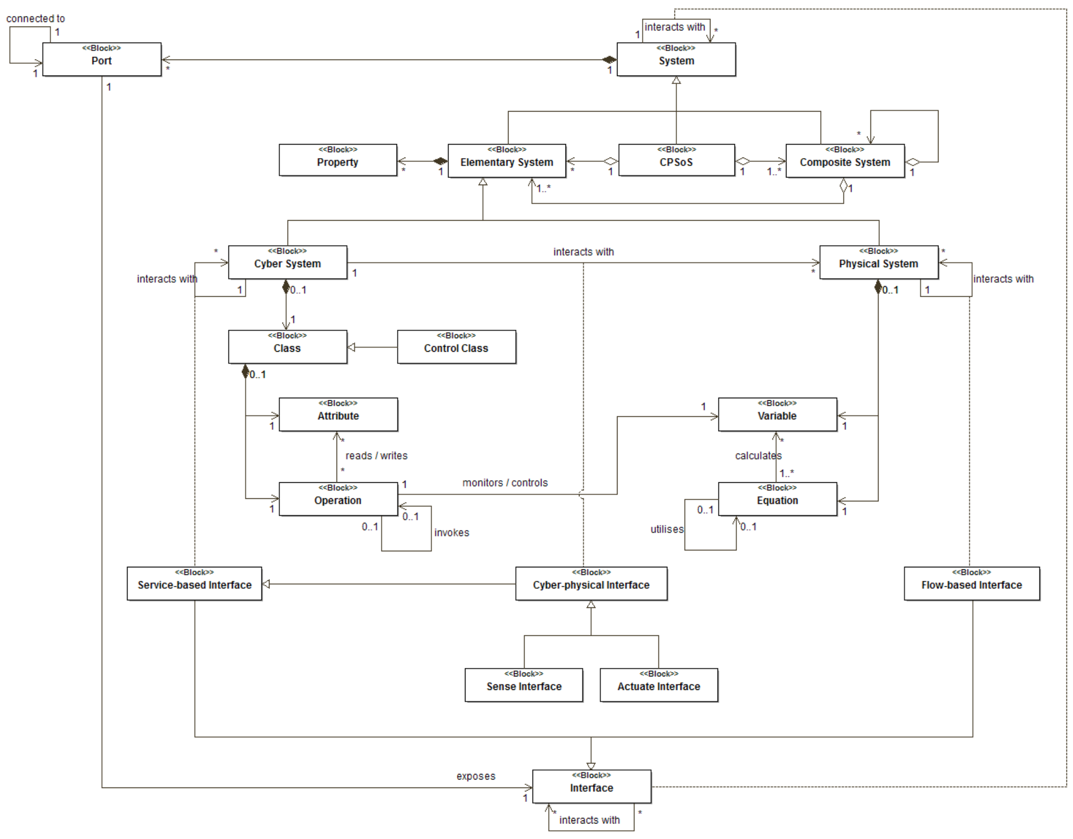


Figure 4.2: Ontology Definition View for CPSoS-AF

4.1.3 Viewpoint Definitions

The specification of an architectural framework requires the definition of a series of viewpoints. Each viewpoint provides a specification for the description of system properties which satisfy particular framework requirements (as described in the Architectural Framework Context View). System properties should be described in terms of CPSoS concepts, where concepts are included in the CPSoS ontology (as described in the Ontology Definition View). Each viewpoint describes a particular perspective of a system architecture, and the combination of all viewpoints should address the complete collection of framework requirements.

The requirements of the proposed architectural framework can be satisfied by the realisation of seven viewpoints:

Composite System Structure Viewpoint supports the identification of constituent systems, including the identification of cyber, physical, and composite systems.

Composite System Connections Viewpoint supports the identification of connection between constituent systems, including the identification of service- and flow-based interfaces of constituent systems.

Logical Composition Viewpoint supports the definition of cyber systems by consideration of their logical structure.

Digital Interfaces Definition Viewpoint supports the definition logical operations for the implementation of digital interfaces, including interfaces facilitating interaction with additional digital components, or the sensing or actuation of physical components.

Controller Behaviour Viewpoint supports the definition of discrete event behaviours by the specification of control logic to be implemented in software.

Equation Definition Viewpoint supports the description of physical phenomena using differential equations.

Equation Utilisation Viewpoint supports the definition of physical systems by the identification of important phenomena and the relationship between system properties and differential equations

By the identification of CPSoS concepts important for the realisation of architectural elements for satisfying particular architectural framework requirements, a series of framework viewpoints are defined. The consideration of particular requirements satisfied by a viewpoint realises the Viewpoint Context View of the CAFF for each defined viewpoint, and the identification of CPSoS concepts important to the realisation of a viewpoint realises the Viewpoint Definition View of the CAFF for each defined viewpoint.

Each Viewpoint Context View can be represented diagrammatically using a Use Case Diagram, as illustrated in Figure 4.3, and each Viewpoint Definition View can be represented using a Block Definition Diagram, as illustrated in Figure 4.4. Diagrams for the full set of defined viewpoints are included in Appendix A.

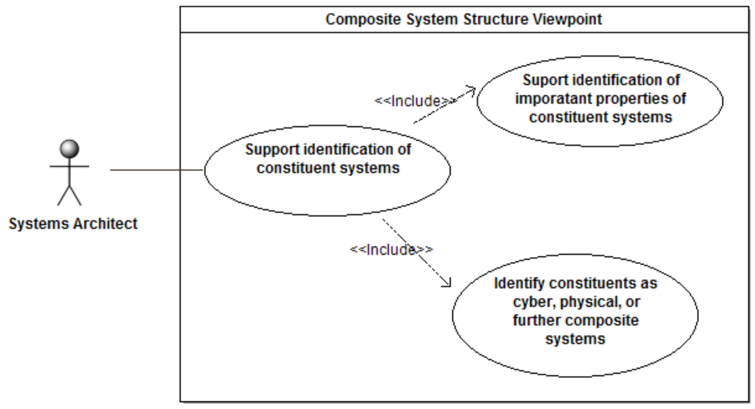

Composite System Structure Viewpoint
Systems Architect
Support identification of constituent systems
<<Include>>
Suport identification of imporatant properties of constituent systems
<<Include>>
Identify constituents as cyber, physical, or further composite systems

Figure 4.3: Viewpoint Context View for Composite System Structure Viewpoint

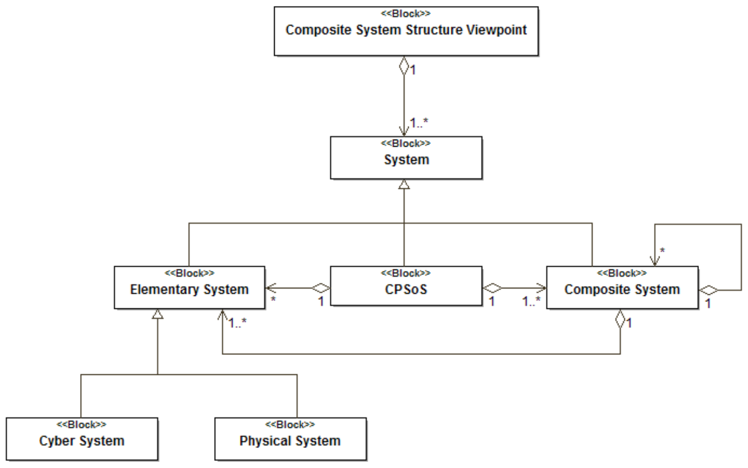

<<Block>> Composite System Structure Viewpoint
1
1..*
<<Block>> System
<<Block>> Elementary System
<<Block>> CPSoS
<<Block>> Composite System
*
1
1
1..*
*
1
1..*
1
<<Block>> Cyber System
<<Block>> Physical System

Figure 4.4: Viewpoint Definition View for Composite System Structure Viewpoint

Viewpoint Relationships

The proposed viewpoints describe alternate perspectives of a system of interest, and information defined in each viewpoint is closely related to information defined in complementary viewpoints.

A Composite System Connections Viewpoint is used to describe the nature of connectivity between elements defined in a Composite System Structure Viewpoint, where the interfaces which enable the connectivity of elements are defined in a Digital Interface Definition Viewpoint.

A Logical Composition Viewpoint is used to define the structure of any discrete event elements included in a Composite System Structure Viewpoint, where the behaviour of logical operations is defined using a Controller Behaviour Viewpoint. Controller Behaviour Viewpoints are also used for the definition of operations defined in a Digital Interfaces Definition Viewpoint.

The behaviour of continuous time elements defined in a Composite System Structure Viewpoint is defined in an Equation Utilisation Viewpoint, which utilises equations defined in an Equation Definition Viewpoint.

The identification of the interconnectivity of viewpoints realises the Viewpoints Relationships View of the CAFF. This can be represented diagrammatically using a Block Definition Diagram, as illustrated in Figure 4.5.

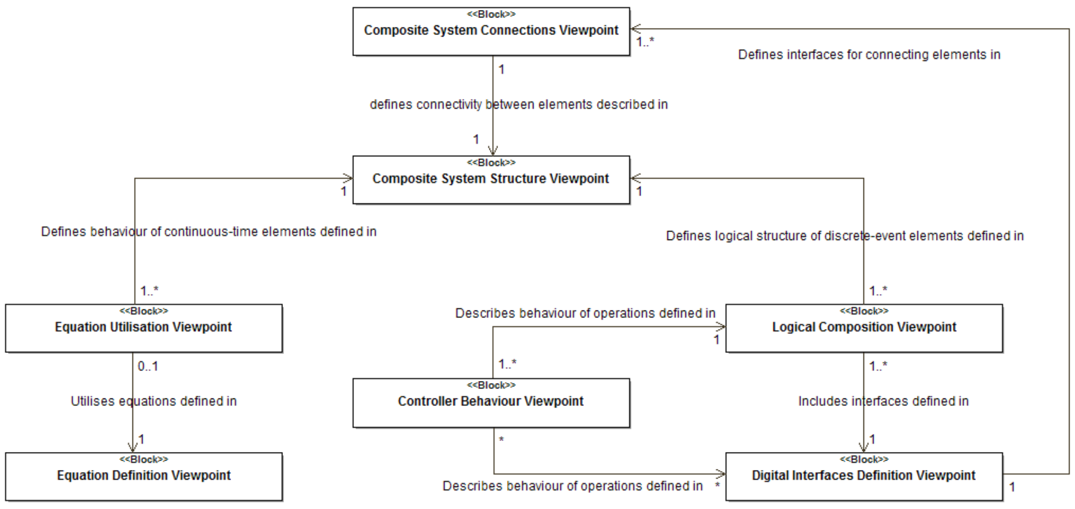

<<Block>>
Composite System Connections Viewpoint
1..*
Defines interfaces for connecting elements in
1
defines connectivity between elements described in
1
<<Block>>
Composite System Structure Viewpoint
1
1
Defines behaviour of continuous-time elements defined in
Defines logical structure of discrete-event elements defined in
1..*
1..*
<<Block>>
Equation Utilisation Viewpoint
Describes behaviour of operations defined in
<<Block>>
Logical Composition Viewpoint
1
0..1
1..*
1..*
<<Block>>
Controller Behaviour Viewpoint
Utilises equations defined in
Includes interfaces defined in
1
*
1
<<Block>>
Equation Definition Viewpoint
Describes behaviour of operations defined in
*
<<Block>>
Digital Interfaces Definition Viewpoint
1

Figure 4.5: Viewpoint Relationships View for CPSoS-AF

4.1.4 Framework Rules

The application of the proposed framework can be further supported by a set of rules constraining the construction of architectural descriptions. Framework rules ensure that system properties described in complementary viewpoints are consistent. The rules for ensuring consistency across the proposed framework include:

CPSoS-AF Rule 1: Each Composite System Structure View (CSSV) must have a corresponding Composite System Connections View (CSCV). Instances of each constituent must be defined and named in the CSCV, and any ports they have must be added.

CPSoS-AF Rule 2: For every *Elementary System* defined in a CSSV, it must be designated as being most appropriately modelled using a DE or CT formalism.

CPSoS-AF Rule 3: For every *Cyber System* defined in a CSSV, a Logical Composition View (LCV) must be defined.

CPSoS-AF Rule 4: For every *Operation* included in a LCV, a Controller Behaviour View (CBV) must be defined.

CPSoS-AF Rule 5: Every LCV must contain exactly one class which implements the *Controller Class* interface. The implementation of this controller interface is a special class which executes a main body periodically. The controller class must implement an operation named 'loop', containing the logic for top-level control of the cyber system.

CPSoS-AF Rule 6: For every *Physical System* defined in a CSSV, an Equation Utilisation View (EUV) must be defined.

CPSoS-AF Rule 7: Any equations used in an EUV must be defined in the Equation Definition Viewpoint (EDV).

CPSoS-AF Rule 8: In an effort to avoid ambiguity, ports defined in a CSCV to facilitate digital interaction must only be used in a single connection. Mulipliple connections utilising a single interface must be facilitated by additional ports.

CPSoS-AF Rule 9: All *Service-based Interfaces* exposed by a port must be defined in the Digital Interfaces Definition View (DIDV).

CPSoS-AF Rule 10: Where a service-based interface is used by a Cyber Constituent to monitor a Physical Constituent, its interface must implement the *Sensor Interface*. Any implementation of a sensor interface must contain a single attribute named 'value', which synchronises a single value with the CT environment. A single operation named 'read' should take no parameters, and return a single real number.

CPSoS-AF Rule 11: Where a service-based interface is used by a Cyber Constituent to control a Physical Constituent, its interface must implement the *Actuator Interface*. Any implementation of an actuate interface must contain a single attribute named 'value', which synchronises a single value with the CT environment. A single operation named 'write' should take a single real number as a parameter, and not offer a return type.

The definition of rules constraining the architectural framework realises the Rules Definition View of the CAFF.

4.2 SysML Profile Definition

The architectural framework for CPSoSs described in Section 4.1 is composed of a collection of viewpoints which ensure the systematic description of CPSoS architectures. The realisation of architectural descriptions using the proposed viewpoints requires the use of a suitable Architectural Description Language (ADL).

This section describes the use of SysML for the realisation of the proposed viewpoints to support the development of an architectural description of a CPSoS. SysML can be tailored for a particular use by the provision of a SysML profile. A SysML profile employs stereotypes to enrich the language with additional features for the expression of concepts specific to a particular domain or application (OMG 2015*a*). Using stereotypes, diagram elements can be specialised to include features for the realisation of a particular viewpoint. Important aspects of a model can be defined as bespoke diagram elements, extending SysML by the addition of tags to specify properties of the new element.

As a profile of UML, each SysML element is related to an underlying UML element. Similarly, each defined stereotype is related to a SysML element or diagram. The definition of a SysML profile describes the relationship between bespoke elements and the elements which they specialise using a series of profile diagrams. By the creation of stereotypes for each viewpoint of the proposed architectural framework, the profile can highlight which viewpoint is represented by each profile diagram. By the creation of stereotypes for CPSoS concepts, they can also be associated with a particular viewpoint diagram.

Section 4.2.1 describes the specialisation of SysML elements for the realisation of the proposed architectural framework. The construction of SysML profile diagrams to relate SysML extensions to underlying elements is described in Section 4.2.2. The complete collection of profile diagrams is summarised in Appendix B.

4.2.1 Profile Overview

Each viewpoint of the proposed architectural framework encapsulates a particular perspective of a given CPSoS, and SysML can be used in the realisation of each viewpoint. SysML diagrams can be specialised for the realisation of particular viewpoints by the introduction of stereotypes.

This section describes the specialisation of candidate SysML elements to facilitate their specialisation for the expression of CPSoS concepts associated with each viewpoint:

Composite System Structure View. The organisation of hierarchical system structures requires mechanisms for the description of component composition, aggregation, and specialisation. These mechanisms are included in *Block Definition Diagrams*, making them an ideal diagram from the realisation of Composite System Structure Views. Realisation of the viewpoint requires the augmentation of native diagram elements to include mechanisms for the designation of nested blocks as either composite or elementary systems, where elementary systems can be further specialised as cyber or physical systems. Elementary systems must also include any properties important to the model, including an attribute name and associated data type.

Composite System Connections View. For each defined Composite System Structure View, a supporting Composite System Connections View must be defined. The definition of system connections is well suited to the use of *Ports* and *Connectors*, included in *Internal Block Diagrams*. For the declaration of specific instances of constituent systems, system properties must be identified and assigned an appropriate value. *Ports* are well suited to identify any exposed interfaces, with *Standard Ports* particularly well suited to the identification of digital interfaces, and *Flow Ports* ideal for the representation of interfaces for the transfer of matter or energy between physical components.

Logical Composition View. The realisation of the logical composition of cyber constituents can be expressed using elements inherited from UML, in particular those elements available in the creation of a *Class Diagram*. Each class diagram must include exactly one control class, which implements the top level control logic for the constituent system.

Digital Interfaces Definition View. Digital interfaces can be expressed using UML *Classes*. Interfaces must be distinguished as communication interfaces facilitating communication between two cyber constituents, or control interfaces facilitating communication between a cyber and physical constituent. Whilst any number of communication interfaces might be defined, control instances must either be of a *Sense Interface* type for describing the reading of data from a physical constituent, or a *Actuate Interface* type for describing the updating of a variable of a physical constituent.

Controller Behaviour View. The realisation of the discrete event behaviours of cyber constituents can be expressed using elements inherited from UML, in particular the elements available in the creation of an *Activity Diagram*.

Equation Definition View. Differential equations describing continuous time phenomena can be recorded as *Constraints* using *Constraint Blocks*, by the realisation of the Equation Definition View using a *Block Definition Diagram*. To ensure that differential equations are compatible with more concrete modelling notations, *Constraint Blocks* should restrict the specification of differential equations to use the SIDOPS+ language for modelling continuous time dynamics (Breunese & Broenink 1997). This is important to support the production of more concrete models which can facilitate simulation of system dynamics.

Equation Utilisation View. The combination of differential equations in the representation of the behaviour of physical constituents can be modelled using *Parametric Diagrams* for the realisation of the Equation Utilisation View. Parametric diagrams can incorporate a collection of *Constraint Blocks*, and *Connectors* for modelling the relationship complementary equation of parameters.

By consideration of the nature of each viewpoint, SysML diagrams appropriate for the realisation of viewpoint are identified, and where necessary extensions to the standard diagrams are required, these are outlined. Table 4.1 summarises the selection of diagram types for the realisation of each viewpoint of the proposed architectural framework.

Table 4.1: Realising Architectural Viewpoints with SysML

Viewpoint	**SysML Diagram Type**
Composite System Structure Viewpoint	Block Definition Diagram
Composite System Connections Viewpoint	Internal Block Diagram
Logical Composition Viewpoint	Block Definition Diagram
Digital Interfaces Definition Viewpoint	Block Definition Diagram
Controller Behaviour Viewpoint	Activity Diagram
Equation Definition Viewpoint	Block Definition Diagram
Equation Utilisation Viewpoint	Parametric Diagram

4.2.2 Profile Diagrams

The constraints defined in Section 4.2.1 can be encapsulated using a series of SysML diagrams. Stereotypes and tags are introduced, and related to the UML metaclasses which have been extended.

Composite Structures

For describing constituent systems of a CPSoS which are themselves a composite of further systems, the proposed architectural framework provides two viewpoints: the Composite System Structure Viewpoint and the Composite System Connections Viewpoint. These viewpoints can be expressed using a Block Definition Diagram (which itself extends the UML Class Diagram), and an Internal Block Diagram (which itself extends the UML Composite Structure Diagram), respectively. Figure 4.6 illustrates how viewpoints associated with describing composite systems can be expressed using SysML. Both of these viewpoints describe a number of *Systems*, so an additional stereotype is created to facilitate the creation of a diagram element for each System. Every instance of a System must be named, so a tag is added to the System stereotype to record this information.

When describing a System, a number of additional assignments can be made to clarify the properties of the system. Figure 4.7 introduces a number of additional stereotypes included in the profile. At the top level, an overall *CPSoS* can be defined using a corresponding stereotype. Within this, a number

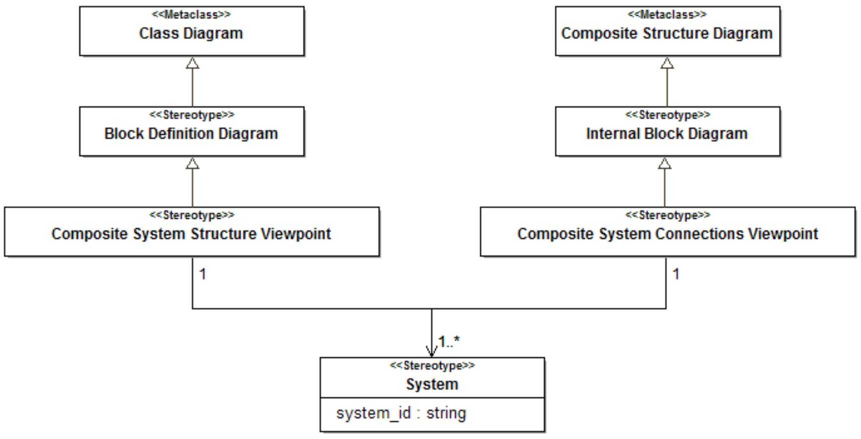


Figure 4.6: Profile Diagram showing viewpoints describing composite structures

of constituent systems can each be classified as either a further *Composite System*, or an *Elementary System*. Further to this, each Elementary System can be designated as a *Cyber Constituent* or a *Physical Constituent*, depending on the type of environment best suited to modelling the system. For each elementary system, a number of *Properties* important to the CPSoS can be identified.

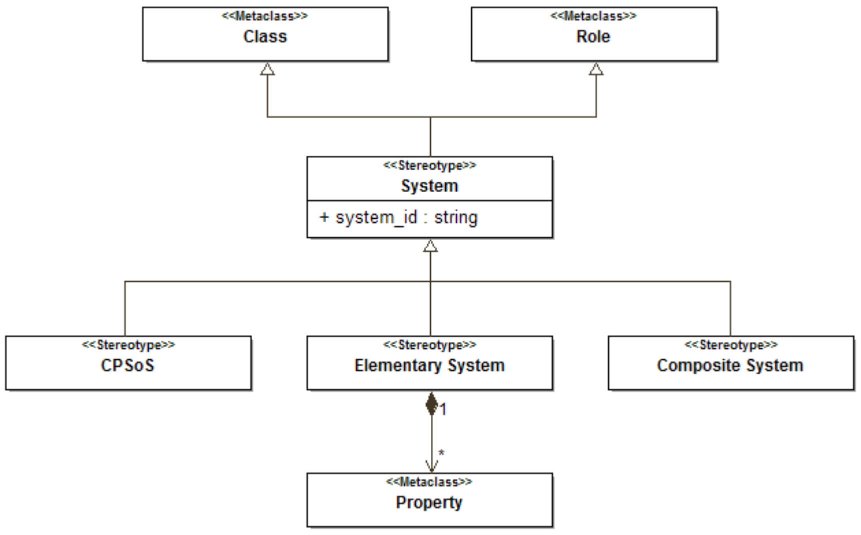


Figure 4.7: Profile Diagram showing elements for describing composite structures

Figure 4.7 also highlights that a system can be defined using complementary mechanisms. In a Composite System Structure View, each System is defined as a Block (extends from UML Class), whereas in a Composite System Connection View, each instance of a System is added as a Part (extends from UML Role).

In a Composite System Connections View, interactions between systems are facilitated by a number of *Interfaces*. Each System can expose an Interface via a *Port*. There are two types of Port for facilitating different types of Interface. A *Standard Port* is used to offer a *Digital Interface*, and a *Flow Port* is used to support the transfer of energy or physical matter. Stereotypes are introduced to enable the addition of each Port type to a Composite System Connections View, both extending the UML Port.

A Cyber Constituent is restricted to only include Standard Ports, whilst Physical Constituents and Composite Systems might offer either Standard or Flow Ports. The Stereotypes for including Ports, as well as the restrictions over the relationships between Port types and System types is included in figure 4.8. Since both types of Port must be named, additional tags are introduced to enforce this.

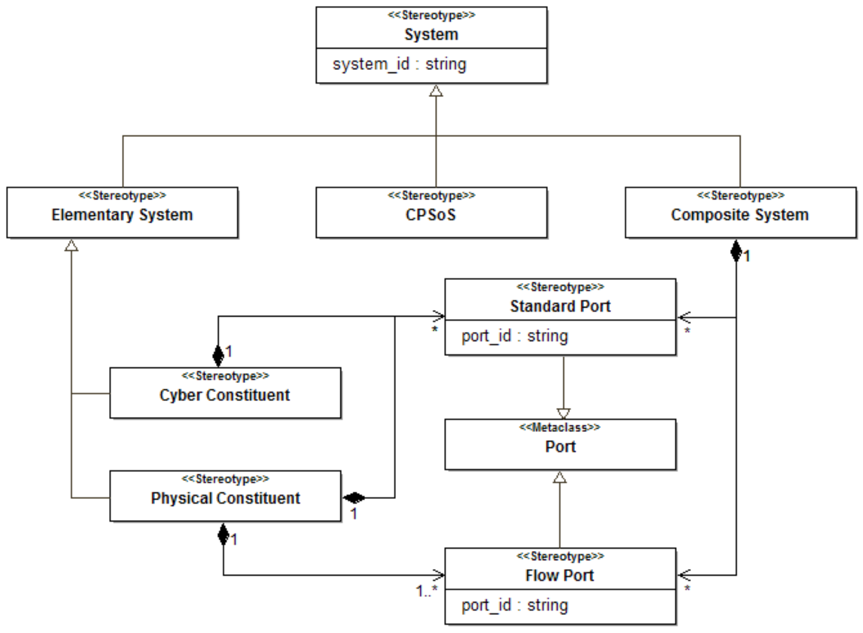


Figure 4.8: Profile Diagram showing elements for describing composite connections

Cyber Constituents

For describing cyber constituent systems of a CPSoS, the proposed architectural framework provides two complementary viewpoints: The Logical Composition Viewpoint and the Cyber Behaviour Viewpoint. The viewpoints can be expressed using a Block Definition Diagram (extended from UML Class Diagram), and an Activity Diagram, respectively. The architectural framework provides an additional viewpoint for defining Digital Interfaces to be used by Cyber Constituents: The Digital Interfaces Definition Viewpoint. This viewpoint can be expressed using an additional Block Definition Diagram.

Figure 4.9 illustrates how viewpoints associated with describing Cyber Constituents and their Interfaces can be expressed using SysML.

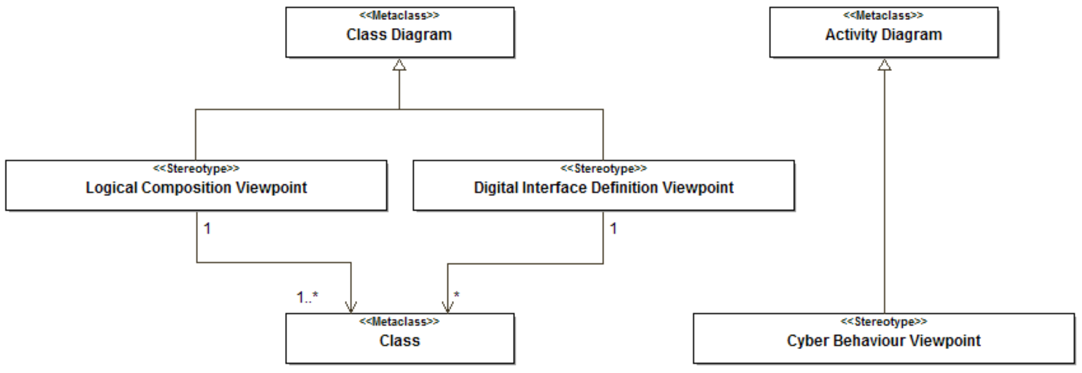


Figure 4.9: Profile Diagram showing viewpoints describing cyber constituents

Each Cyber Constituent must have a single *Controller Class*, which will periodically execute a main control sequence (this must be defined as an operation named ‘Loop’). A stereotype is added to the profile for facilitating the specification of a Controller Class, which is a specialised UML Class, and may utilise a number of other Classes.

Figure 4.10 highlights additional elements to support the specification of the logical composition of a Cyber Constituent.

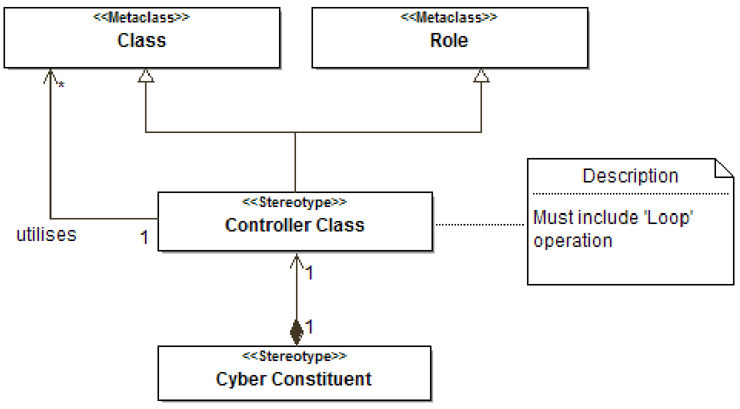
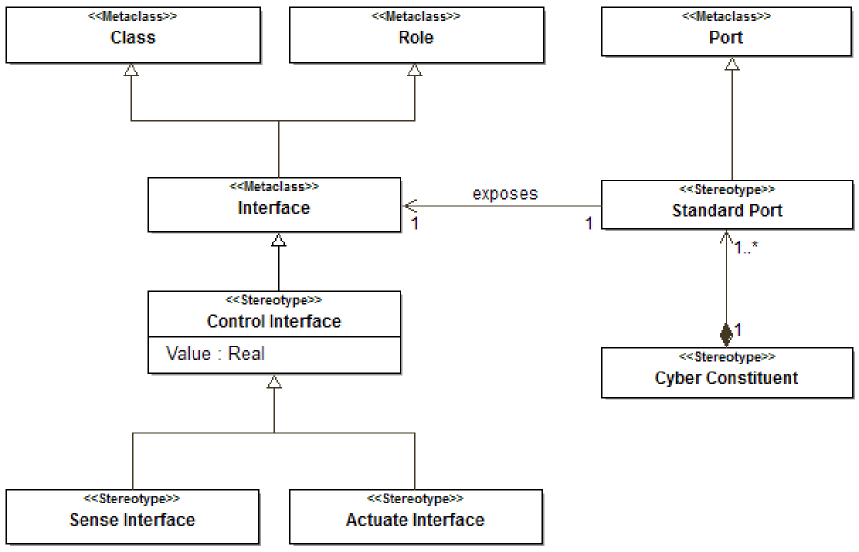


Figure 4.10: Profile Diagram showing elements describing cyber structures

A number of Interfaces are defined to facilitate Cyber Constituent interaction (with either other Cyber Constituents, or with Physical Constituents). Each Interface can be described using SysML by the definition of an Interface Block. For facilitating interaction between cyber and physical elements, a specialised *Control Interface* can be defined. A Control Interface contains details of the value which is to be synchronised between DE and CT models, and can either pass the value from CT to DE via a *Sense Interface* or from DE to CT via an *Actuate Interface*. Figure 4.11 highlights additional elements to support the specification of digital interfaces.



<<Metaclass>> Class
<<Metaclass>> Role
<<Metaclass>> Port
<<Metaclass>> Interface
exposes
1
1
<<Stereotype>> Standard Port
1..*
1
<<Stereotype>> Control Interface
Value : Real
<<Stereotype>> Cyber Constituent
<<Stereotype>> Sense Interface
<<Stereotype>> Actuate Interface

Figure 4.11: Profile Diagram showing elements describing digital interfaces

Physical Constituents

For describing constituent systems of a CPSoS which are to be modelled in a CT formalism, the proposed architectural framework provides two viewpoints: The Equation Definition Viewpoint and the Equation Utilisation Viewpoint. These viewpoints can be expressed using a Constraints Diagram (extended from BDD, which is extended from the UML Class Diagram), and a Parametric Diagram (extended from the UML Composite Structure Diagram), respectively. Figure 4.12 illustrates how viewpoints associated with describing Physical Constituents can be expressed using SysML. An Equation Definition View defines a number of *Constraint Blocks*, which are also utilised by each Equation Utilisation View, so a stereotype is included for adding Constraint Blocks to each viewpoint.

Each Constraint Block includes a *Constraint*. This is used to express some physical phenomena as a differential equation using the SIDOPS+ language. Constraints can be linked linked to a number of other Constraints, as well as to constants recorded as a Property of the system. This is illustrated in Figure 4.13.

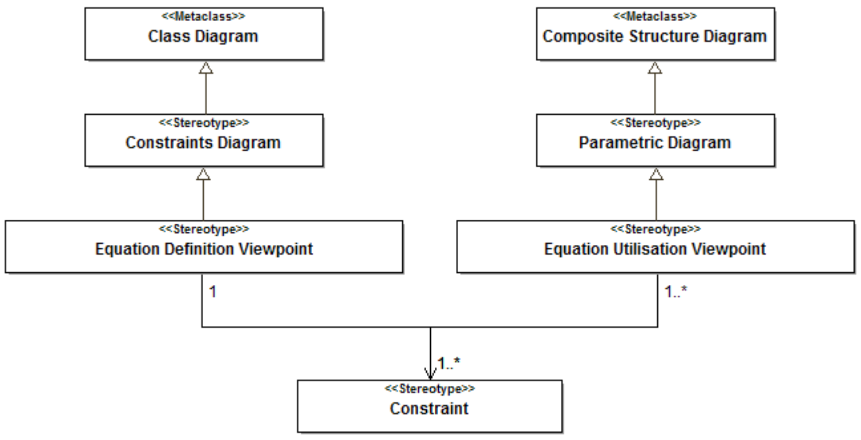


Figure 4.12: Profile Diagram showing viewpoints describing physical constituents

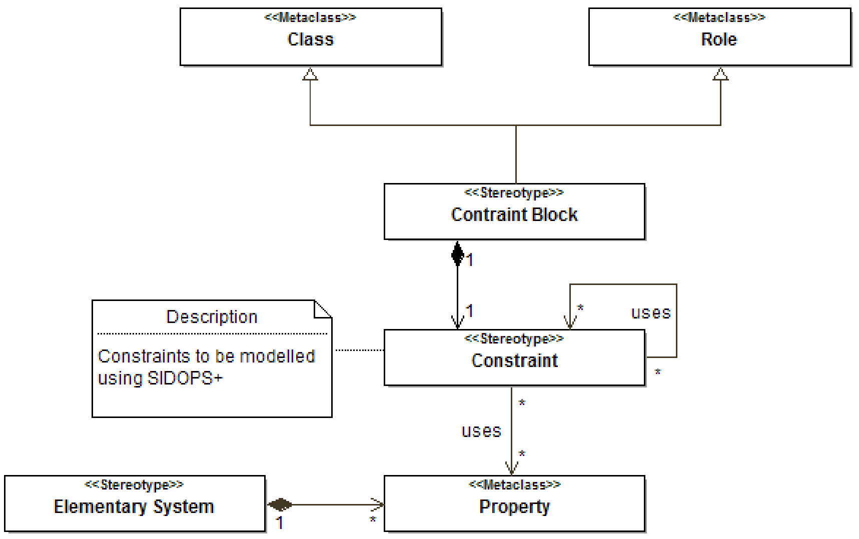


Figure 4.13: Profile Diagram showing elements for describing CT behaviour

4.3 Summary

This chapter describes the development of an architectural framework for supporting the description of architectural descriptions of CPSoSs, and a SysML profile for guiding the realisation of the proposed framework. The framework is developed by the application of the CAFF, a meta-framework for the specification of architectural frameworks.

The proposed framework is defined by the realisation of the seven views of the CAFF. An Architectural Framework Context View ensures that the requirements of the framework are understood, by consideration of the needs of any relevant stakeholders. Terms and concepts related to CPSoS engineering are defined in an ontology, and used to define a collection of viewpoints for satisfying the framework requirements.

A SysML profile extends SysML, which itself extends UML. The profile tailors the use of SysML for the realisation of CPSoS architectures by extending the language by the addition of domain-specific constructs. A series of profile diagrams illustrate the relationship between relationship between elements defined in UML, extensions of UML elements defined in SysML, and profile elements.

Realisation of the proposed architectural frameworks using the bespoke SysML profile for the description of CPSoS architectures ensures that architectural descriptions are sufficiently detailed to support the exploitation or the model in the development of more concrete representations.

5 Co-model Generation

An architectural description and supporting SysML profile to support the engineering of CPSoS are defined in Chapter 4. Whilst architectural descriptions are a useful tool for the organisation of complex architectures, they do not facilitate exploration of emerging behaviours. Analysis of properties and behaviours which emerge as a result of interaction between both constituent systems and between digital and physical elements requires the provision of executable models. Heterogeneous co-models enable the inclusion of rich descriptions of both discrete event and continuous time phenomena by the coupling of VDM and 20-sim modelling environments.

Whilst complementary architectural descriptions and co-models of a system provide independently useful mechanisms for the model-based engineering of CPSoSs, the information contained within each model should be consistent. To ensure consistency between architectural and executable models, a systematic approach to the production of co-models based on architectural descriptions is essential.

This chapter provides a specification for the automated mapping of information contained within an architectural description to a corresponding co-model. By automating the generation of co-models, a mapping can be defined to ensure consistency between complementary abstractions. To enable the generation of co-models, architectural descriptions must be sufficiently detailed to include any information important in enabling co-simulation of a system. Ensuring sufficiently detailed architectural descriptions is facilitated by the use of the proposed architectural framework.

The specification of a mapping between complementary modelling formalisms requires sufficiently abstract representation of each language. To generate a co-model composed of 20-sim and VDM models based on a SysML specification, an abstract representation of the underlying meta-models of each formalism must be defined. These meta-models are defined in Section 5.1. With each of the languages modelled, a function can be defined which maps information between them. This mapping function is defined in Section 5.2.

5.1 Abstract Model Specification

The production of abstract representation of computational languages can be facilitated by the use of a *specification language*. A specification language is a highly abstract mechanism for the high-level description of systems. More abstract than programming languages, abstractions constructed using a specification language exclude consideration of implementation details, and describe what a system should do rather than how it should be done.

One of the longest established and widely used specification languages used in the description of computational languages is VDM-SL (Bekič & Jones 1984). VDM-SL enables the modelling of computing systems (including languages) and has been used extensively in the engineering of language compilers. VDM-SL is an ideal candidate for the description of functional specifications of systems. Data is defined in terms of basic data types which can be constrained according to requirements, and can be included in collections such as sets, sequences and mappings. Functionality is modelled in terms of functions and operations over the data types. The VDM-SL notation is standardised in ISO/IEC 13817 (1996).

VDM-SL is used to define an abstract representation of SysML models in Section 5.1.1. Since the mapping of architectural descriptions is dependent on the use of the proposed SysML profile, only language constructs included in the profile are modelled (including defined extensions to standard SysML). VDM-RT and 20-sim features included in co-models are also abstractly defined using VDM-SL in Section 5.1.2. A complete specification of the abstract syntax describing both SysML models and co-models is given in Appendix C.

5.1.1 SysML Profile

Architectural models realised using proposed SysML profile are composed of a number of views. Whilst each view isolates particular concepts or components for a particular purpose, the information described across an entire collection of views is encapsulated in an underlying meta-model. It is this meta-model which is described using VDM-SL.

Description of a SysML meta-model must include consideration of the structuring of model elements, the interfaces exposed by each element, and the connections between elements. Additionally, mechanisms for the representation of both logical structure and physical behaviours are required.

Model Structure

The most primitive element for the representation of a system element in a SysML model is a *block*. These elements are explicitly included in the abstract representation by the definition of a $Block$ structure. Each *block* might expose a number of *interfaces* by the inclusion of a collection of *ports*, and the composition of a block might be a composite of further blocks, or a representation of a cyber or physical constituent system. Blocks composed of nested elements are represented by the provision of a $CompositeContent$ structure, which records the nested elements and the relationships between them.

$$SysMLModel = CompositeContent$$

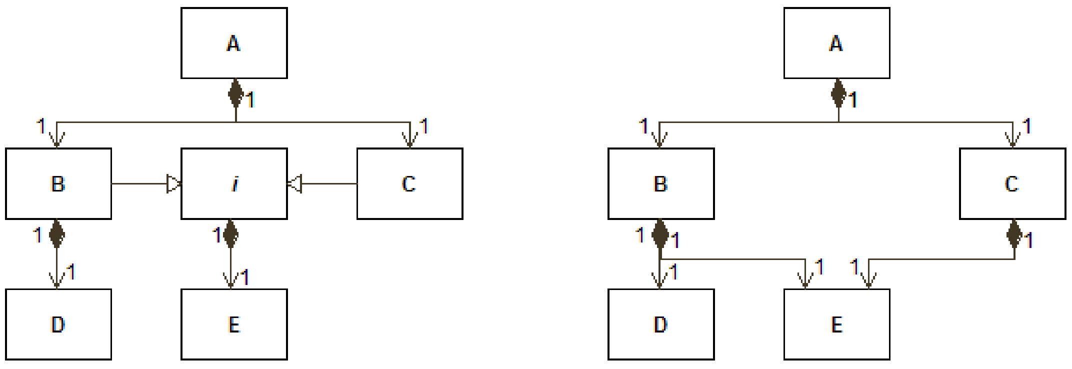
$$\begin{aligned} CompositeContent \;::\; \quad children &: Id \xrightarrow{m} Block \\ connections &: Flow\text{-}\mathbf{set} \end{aligned}$$

$$\begin{aligned} Block \;::\; \quad ports &: Id \xrightarrow{m} PortType \\ contents &: CompositeContent \mid NodeDescription \end{aligned}$$

$$NodeDescription = PhysicalNode \mid CyberNode$$

This structure encapsulates only the information necessary for the generation of a co-model, and does not preserve additional information which do not directly correspond to a co-model element or property. Specifically, SysML constructs such as aggregation and specialisation are not represented in the VDM-SL abstraction. The impact of this is that a range of disparate SysML models might correspond to the same abstract representation by the omission of detail. Resultingly, whilst the given abstract of SysML is sufficiently rich to enable the generation of a co-model, it does not contain necessary detail to facilitate translation back to SysML. Whilst it is outside of the scope of this work to support a two-way translation process, it would be possible with a richer abstract syntax. This would require the addition of a header to a resultant co-model, containing meta-data redundant to the co-model, but essential for the regeneration of a complete SysML model.

An example of two contrasting structures which would share identical representations in the abstract model is illustrated in Figure 5.1. In Figure 5.1(a), SysML specialisation is used to describe how two elements (A and B) are implementations of some generic element (i). Encoding this structure using the proposed VDM-SL constructs would result in a description identical to an encoding of the structure outlined in Figure 5.1(b):



(a) Original Structure (b) Possible Alternative

Figure 5.1: VDM-SL Abstraction of SysML Structural Elements

let $BBody = mk\text{-}CompositeContent(\{\text{'D'} \mapsto mk\text{-}Block(\text{-},\text{-}), \text{'E'} \mapsto mk\text{-}Block(\text{-},\text{-})\}, \text{-})$ **in**
let $CBody = mk\text{-}CompositeContent(\{\text{'E'} \mapsto mk\text{-}Block(\text{-},\text{-})\}, \text{-})$ **in**
$\text{'A'} = mk\text{-}CompositeContent(\{\text{'B'} \mapsto CBody, \text{'C'} \mapsto DBody\}, \text{-})$

Modelling Interfaces

Blocks can interact via a specified interface. The interface of each block can contain a number of ports, each addressable by a unique identifier. SysML facilitates the specification of block interfaces using service ports and flow ports. Each port type explicitly included in the abstract representation.

$PortType = FlowPort \mid ServicePort$

Where an interface between two elements represents some sort of transfer of a physical nature, SysML offers flow ports to represent interface properties. Flow ports are used to specify the type of material which the port can convey and the direction which the material can pass through the port. The particular material which the port conveys is not represented in the abstract syntax since the generated co-model employs a generic domain for all physical transfers. Materials can flow in to a port, out of a port, or both in and out of a port.

$FlowPort :: direction : Direction$

$Direction = \text{IN} \mid \text{OUT} \mid \text{BOTH}$

Service ports are commonly used for the specification of an interface between software systems, where they expose some digital interface for interaction between blocks. When modelling systems of a cyber-physical nature, service ports are employed not only to represent interaction between two discrete elements, but also to represent interaction between a discrete element and a continuous element. To prevent ambiguity in the use of SysML ports, the proposed SysML profile restricts service ports to expose a maximum of one interface.

$ServicePort :: exposes : Interface$

The properties of an interface are determined by its purpose. An interface can be used to facilitate the exchange of information between two controllers, or between a control element and a plant element. These two types of interface are represented by a $CommunicationInterface$, a $ControlInterface$, respectively. An interface is defined as a named operation.

$Interface = CommunicationInterface \mid ControlInterface$

Where an interface is used to exchange information between two control elements, the interface is defined as an operation with any number of parameters, and a possible return type.

$$\begin{aligned} CommunicationInterface :: interfaceName &: Id \\ parameters &: Id \xrightarrow{m} UnitType \\ returns &: [UnitType] \end{aligned}$$

$UnitType$ can be used to represent one of several permitted types. To ensure compatibility with both VDM-RT and 20-sim, only units common to both formalisms are permitted, including real numbers, integers, and boolean values.

$UnitType = \text{REAL} \mid \text{INTEGER} \mid \text{BOOLEAN} \mid \ldots$

Where an interface is used to exchange information between a control element and a plant element, again an interface is defined as an operation. These control interfaces are modelled as specialised communication interfaces, with constraints to restrict the use of parameters and return fields. A controller might utilise a control interface to gather data from some sensor, or to update the a value of some actuator.

$ControlInterface = SenseInterface \mid ActuateInterface$

Where a control interface describes a controller sensing its environment, the interface operation takes no parameters and returns a single Real number.

$SenseInterface = CommunicationInterface$

where

$inv\text{-}SenseInterface(\text{-}, p, r) \quad \triangleq \quad p = \{\,\} \wedge r = \text{REAL}$

Where a control interface describes a controller actuating its environment, the interface operation takes a single Real number to be applied to the continuous model via the connected port, and does not facilitate a return value.

$ActuateInterface = CommunicationInterface$

where

$inv\text{-}ActuateInterface(\text{-}, p, r) \quad \triangleq \quad \mathbf{let}\ x \in \mathbf{rng}\, p\ \ \mathbf{in}$
$\qquad x = \text{REAL} \wedge \mathbf{card}\,\mathbf{dom}\, p = 1 \wedge r = \mathbf{nil}$

Connecting Elements

Block ports can be connected by a number of different *flows*. *Item flows* are used when a connection represents the transfer of a physical material, whilst *information flows* are used when a connection represents the transfer of some data.

$Flow = ItemFlow \mid InformationFlow$

An $ItemFlow$ construct is used for the specification of the particular material being transferred, and the source and target ports of the transfer. Since a generic domain is used in the resultant co-model, the material properties are not recorded.

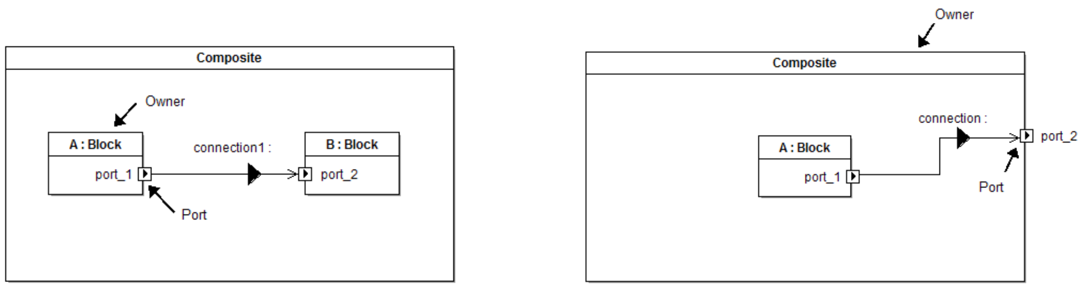
$$\begin{aligned} ItemFlow ::\ \ & name : Id \\ & source : PortReference \\ & target : PortReference \end{aligned}$$

An *InformationFlow* construct is used for the specification of the interface implemented by a connection, as well as which element provides the interface and which element requires the interface.

$$\begin{aligned} \mathit{InformationFlow} \;::\; \mathit{name} &: \mathit{Id} \\ \mathit{implements} &: \mathit{Interface} \\ \mathit{providedBy} &: \mathit{PortReference} \\ \mathit{requiredBy} &: \mathit{PortReference} \end{aligned}$$

Connections identify ports using a *PortReference*. A port reference contains the name of the child block offering the port as part of its interface (the 'owner'), and the unique identifier of the port within the interface of the owner, as illustrated in Figure. 5.2(a). Where a connection is resolved to an external port of the containing block rather than a port of a child element, the owner is omitted, as illustrated in Figure. 5.2(b).

$$\begin{aligned} \mathit{PortReference} \;::\; \mathit{owner} &: [\mathit{Id}] \\ \mathit{port} &: \mathit{Id} \end{aligned}$$



(a) Composite content port referencing (b) Owner port referencing

Figure 5.2: Composite content port referencing

Physical Behaviour

Continuous time phenomena are described using the SysML profile by the linking of equations described by *Constraint Blocks*. For each *PhysicalNode*, continuous time equations used to model its behaviour are defined, alongside any declared constants used by equations, and bindings between values. The SysML profile ensures that any equations are declared in SIDOPS+, the language used to model continuous time equations in 20-sim.

$$\begin{aligned} \mathit{PhysicalNode} \;::\; \mathit{constraints} &: \mathit{Id} \xrightarrow{m} \mathit{Constraint} \\ \mathit{constants} &: \mathit{Id} \xrightarrow{m} \mathit{Constant} \\ \mathit{bindings} &: \mathit{Binding}\text{-}\mathbf{set} \end{aligned}$$

A $Constraint$ outlines the structure of a continuous time equation, as well as details of any parameters used in the equation, whilst a $Constant$ includes a data type and assigned value, where $ValueTypes$ includes any valid instance of a $UnitType$.

$$\begin{aligned} Constraint :: \quad body &: Equation \\ parameters &: Id \xrightarrow{m} UnitType \end{aligned}$$

$$\begin{aligned} Constant :: \quad value &: ValueType \\ unit &: UnitType \end{aligned}$$

$$ValueType = \mathbb{Z} \mid \mathbb{R} \mid \mathbb{B} \ldots$$

A $Binding$ represents a value that is accessed by more than one equation. A value can be declared as a constant, or be maintained as a parameter of an equation. A binding may also indicate that a model element exposes a value to an externally facing port. Where a binding declares the value to be shared, all participants of the binding will maintain the same value. A binding may be named to aid clarity in the resulting co-model, but it is not mandatory.

$$\begin{aligned} Binding :: \quad id &: [Id] \\ participants &: ValueRef\textbf{-set} \end{aligned}$$

$$ValueRef = ParameterRef \mid ConstantId \mid PortId$$

$$\begin{aligned} ParameterRef :: \quad constraint &: Id \\ parameter &: Id \end{aligned}$$

$$ConstantId = Id$$

$$PortId = Id$$

Logical Structure

Realisation of the proposed architectural framework includes description of the logical structure and intended behaviour of digital elements using mechanisms inherited from UML. The abstract representation of the system architecture must include additional mechanisms for the inclusion of these properties to support the translation from UML to VDM-RT models.

The generation of software descriptions from UML constructs is well exercised, and a mapping between UML Class Diagrams and Activity Diagrams and VDM-RT is specified by Lausdahl, Lintrup & Larsen (2008). Since this mapping uses VDM-SL to describe a specification for the mapping of UML to

VDM-RT by abstract representation of the meta-models underpinning each formalism, this work does not duplicate this description. Instead, this work complements the existing mapping specification by the definition of additional mechanisms to facilitate the embedding of VDM-RT in a co-model. A complete implementation of the model mapping specification must include the specification presented here alongside the specification defined by Lausdahl et al. (2008).

To enable VDM-RT constructs to be linked to a co-model, a $\mathit{CyberNode}$ provides a distinction between a $\mathit{ControllerClass}$ and an $\mathit{AuxClass}$. Each $\mathit{CyberNode}$ must implement a $\mathit{ControllerClass}$, containing a primary control loop, but might utilise any number of $\mathit{AuxClasses}$ in its specification. The inclusion of the $\mathit{AuxClass}$ mecahnisms provides a placeholder for the embedding of structures defined by Lausdahl et al. (2008).

$\mathit{CyberNode} = \mathit{Class}\text{-}\mathbf{set}$

$\mathit{Class} = \mathit{ControllerClass} \mid \mathit{AuxClass}$

$\mathit{ControllerClass} = \mathit{AuxClass}$

$$\begin{array}{lrcl} \mathit{AuxClass} :: & \mathit{className} & : & \mathit{Id} \\ & \mathit{classBody} & : & \ldots \end{array}$$

5.1.2 Co-Models

A Crescendo co-model is a composite of a VDM-RT model describing discrete event systems implemented in software, and a 20-sim model describing the continuous dynamics of physical systems. These two component models are included in the abstract representation of co-models.

$$\begin{array}{lrcl} \mathit{CoModel} :: & \mathit{de\text{-}model} & : & \mathit{DEModel} \\ & \mathit{ct\text{-}model} & : & \mathit{CTModel} \end{array}$$

VDM-RT

Whilst this work does not include constructs to replicate the mapping between UML and VDM-RT by Lausdahl et al. (2008), constructs specific to the participation of VDM-RT models in a co-model are included in the meta-model abstraction.

To facilitate the inclusion of a VDM-RT model in a co-model, a representation of a DE model must include consideration of distinct computational devices and any communication between them. A System

class defines a number of virtual CPUs to be utilised by the DE environment. As well as these virtual execution environments, a number of virtual busses can be defined to facilitate communication between CPUs. A bus can link any number of CPUs, and providing that a pair of CPUs are connected by some bus then processes running on one CPU can interact with processes running on another CPU.

A $\mathit{Contract}$ describes any relationships between elements modelled in the DE environment and those modelled in the CT environment. By their inclusion in a contract, the co-simulation engine is able to synchronise values between the two environments.

$$\begin{aligned} \mathit{DEModel} :: \quad \mathit{system} &: \mathit{VDMSystem} \\ \mathit{contract} &: \mathit{Contract} \end{aligned}$$

$$\begin{aligned} \mathit{VDMSystem} :: \quad \mathit{cpus} &: \mathit{Id} \xrightarrow{m} \mathit{CPU} \\ \mathit{busses} &: \mathit{Bus}\text{-}\mathbf{set} \end{aligned}$$

Each virtual CPU is used to deploy a controller, defined by a VDM-RT class. Each Controller class must implement a control loop, which is executed periodically throughout a co-simulation. CPUs can specify real-time constraints over the execution of digital constituents, such as execution capacity (instructions per second) and the frequency of periodic execution. The proposed architectural framework does not include consideration of computational latency or execution frequency, but mechanisms for the representation of timing constraints is a possible extension to the baseline framework to better facilitate time-critical computational systems.

$$\mathit{CPU} :: \mathit{controller} : \mathit{VDMController}$$

To enable the proper allocation of controllers to virtual execution environments, each controller is uniquely identifiable. A control can utilise a collection of *Sensor* and *Actuator* classes to read and write values across the DE/CT boundary. To facilitate co-simulator access to sensor and actuator classes, they must be defined in the *System* and referenced by controllers.

$$\begin{aligned} \mathit{VDMController} :: \quad \mathit{implementation} &: \mathit{Id} \\ \mathit{interfaces} &: \mathit{Id} \xrightarrow{m} \mathit{Id} \end{aligned}$$

To facilitate communication between controllers deployed in distinct execution environments, a virtual *Bus* is defined. The definition of a Bus includes reference to any CPUs which can communicate on it. Similar to CPUs, virtual busses can be extended to model performance constraints, but this is not accommodated by their proposed representation.

$$\mathit{Bus} = \mathit{Id}\text{-}\mathbf{set}$$

A co-simulation contract is essential in enabling synchronisation of DE and CT environments. A contract explicitly links variable names in DE and CT models. The contract also expresses the direction which a value is passed between DE and CT environments: a CT value sensed by the controller is described as a $\mathit{Monitored}$ variable, whilst a value actuated by a controller is described as a $\mathit{Controlled}$ variable. The communication of data between cyber and physical models is constrained to real numbers. Whilst this is restrictive, further data types (e.g. Boolean) can be encoded as reals for their communication between models.

$\mathit{Contract} = \mathit{Id} \xrightarrow{m} \mathit{SharedVariable}$

$\mathit{SharedVariable} = \textsc{monitored} \mid \textsc{controlled}$

20-Sim

An abstract representation of a 20-sim model must include consideration of the hierarchical structuring of element, element interfaces and communication between elements, and the description of physical phenomena using differential equations.

A 20-sim model is a composite of a collection of *Submodels*. A submodel might be composed of a collection of further submodels, or describe a series of differential equations. Each submodel is represented as a $\mathit{CompositeModel}$ or an $\mathit{ElementaryModel}$, and includes an interface composed of a collection of $\mathit{SubmodelPort}$. $\mathit{Connections}$ are used to represent the transfer of data between ports.

$\mathit{CTModel} = \mathit{CompositeModel}$

$$\begin{aligned} \mathit{CompositeModel} ::\quad \mathit{elements} &: \mathit{Id} \xrightarrow{m} \mathit{Submodel} \\ \mathit{connections} &: \mathit{Connection}\textbf{-set} \end{aligned}$$

$$\begin{aligned} \mathit{Submodel} ::\ \mathit{modelType} &: \mathit{CompositeModel} \mid \mathit{ElementaryModel} \\ \mathit{interface} &: \mathit{Id} \xrightarrow{m} \mathit{SubmodelPort} \end{aligned}$$

Submodel interfaces can contain a number of ports. 20-sim supports the specification of two types of port: signal ports and bond ports. All ports are assigned an orientation of either input or output.

20-sim supports the definition of *Signal Ports* and *Bond Ports*. Whilst signal ports expose a single named variable, bond ports are more specialised and are characterised by a pair of linked variables. Signal Ports require the explicit definition of an orientation, whilst Bond ports can be bidirectional, with either side of a connection able to update a common variable. The initial orientation of a bond port is determined by connections to it, and bidirectional ports in SysML can be used to indicate where a bond port is appropriate. Bond ports offer a powerful shorthand for modelling the physical dynamics, though their functionality can be emulated by equations linking a pair of signal ports.

$$\begin{aligned} \mathit{SubmodelPort} :: \mathit{orientation} &: \mathit{Orientation} \\ \mathit{type} &: \text{SIGNAL} \mid \text{BOND} \end{aligned}$$

$$\mathit{Orientation} = \text{INPUT} \mid \text{OUTPUT}$$

Values can be passed between two submodels, or between a submodel and the interface of its parent model. 20-sim supports the specification of two types of connection: bonds and signals. The use of bonds is restricted to connections between two bond ports, and similarly the use of signals is restricted to connections between two signal ports. Port referencing of 20-sim ports is implemented similarly to the port referencing mechanisms proposed for SysML connections.

$$\begin{aligned} \mathit{Connection} :: \mathit{name} &: [\mathit{Id}] \\ \mathit{source} &: \mathit{PortReference} \\ \mathit{target} &: \mathit{PortReference} \end{aligned}$$

Elementary submodels contain a series of equations, either for the description of system dynamics, or for linking 20-sim variables to a DE model. To facilitate both types of equation-based model, the abstract representation includes $\mathit{PhysicalModels}$ and $\mathit{ControlLinkModels}$.

For modelling physical dynamics, equations defined in SysML Constraints are included in an $\mathit{equation}$ block in an elementary 20-sim model. A $\mathit{ValueMap}$ is used to maintain values used or updated by equations. Two types of value are maintained: $\mathit{Paramters}$ describe static values associated with some property of the system, and $\mathit{Variable}$ values which describe variables used between multiple equations.

$$\mathit{ElementaryModel} = \mathit{PhysicalModel} \mid \mathit{ControlLinkModel}$$

$$\begin{aligned} \mathit{PhysicalModel} :: \mathit{valueMap} &: \mathit{Id} \xrightarrow{m} \mathit{ValueInfo} \\ \mathit{equations} &: \mathit{Equation}\text{-}\mathbf{set} \end{aligned}$$

$$\mathit{ValueInfo} = \mathit{Parameter} \mid \mathit{Variable}$$

$$\begin{aligned} \mathit{Parameter} :: \mathit{value} &: \mathit{valueType} \\ \mathit{unit} &: \mathit{UnitType} \end{aligned}$$

$$\mathit{Variable} :: \mathit{unit} : \mathit{UnitType}$$

Where 20-sim variables are monitored or controlled by a DE model, specialised equation models link local variables to variables in the co-simulation contract. This is facilitated in the abstract representation by a collection of $\mathit{Externals}$, which identify whether a variable is monitored by a DE model (Export variables) or controlled by a DE model (Import variables).

$\mathit{ControlLinkModel} = \mathit{Id} \xrightarrow{m} \mathit{External}$

$\mathit{External} = \textsc{import} \mid \textsc{export}$

5.2 Meta-Model Mapping

This section describes a process for generating co-models based on a SysML specification. The transformation process uses an abstract representation of both SysML and co-models, defined using the syntax described in Section 5.1. By employing the architectural framework and the supporting SysML profile defined in Chapter 4 to guide the production of an architectural model of a CPSoS, the abstract model will be sufficiently rich to support the translation to a co-model. Section 5.2.1 outlines the basic steps required to complete the transformation process. Section 5.2.2 presents the encoding of these steps into a number of translations expressed as VDM-SL functions, where each function takes some part of the SysML abstract syntax and returns a part of the co-model abstract syntax. A complete specification of the translation functions is given in Appendix D.

5.2.1 Transformation Outline

There are a number of stages to completing the transformation from SysML to co-model for a given CPSoS. Firstly, composite systems must be decomposed into their consituent parts to identify an appropriate target domain. Constituent systems identified as being CT are added to a 20-sim model, whilst those identified as being DE are modelled using VDM-RT. A collection of candidate CT constituents are structured using 20-sim sub-models, whilst a collection of DE consitituents are modelled using virtual processors.

By inspecting the interations between constituents, the models can be connected. Physical connections are modelled in 20-sim, whilst digital interfaces are modelled in VDM-RT. Where digital elements interact with other digital elements, a number of virtual busses connect the virtual processors to enable necessary communications. Where digital elements interact with physical constituents, a co-simulation contract facilitates the coupling of the DE and CT models.

The transformation process is split into two parts: generating a 20-sim model, and generating a VDM-RT model (the VDM-RT model includes the co-simulation contract). For each of these models, there are a number of steps to completing its generation from SysML.

5.2.2 Transformation Implementation

This section describes a translation function for generating a co-model based on a SysML specification, using the abstract syntax defined in Section 5.1.

A single function takes a SysML model as a parameter, and returns a corresponding co-model. This function is defined as:

$$GenerateCoModel : SysMLModel \to CoModel$$

$$GenerateCoModel(sysml) \quad \triangleq$$
$$\quad mk\text{-}CoModel(GenerateCTModel(sysml), GenerateDEModel(sysml))$$

This function is dependent on a number of other functions which are explained in the following sections.

Firstly, a 20-sim model is created, containing all structural elements of the constituent systems. A submodel is generated for each constituent, with further submodels created for any composite elements. Interfaces are generated for each submodel, and physical connections between submodels are made.

For each elementary submodel created in 20-sim, a series of equations specifying CT behaviour is generated. Where submodels correspond to DE control elements, the necessary connections to a DE environment are made.

A VDM-RT model is also generated according to SysML specification. VDM-RT classes are generated according the translation functions specified in section.

Structural Elements

The root of the SysML model is a *CompositeModel*, which is translated in the same way as any composite submodel, only with the restriction that it cannot offer an interface:

$$\mathit{GenerateCTModel} : \mathit{CompositeContent} \to \mathit{CTModel}$$

$$\mathit{GenerateCTModel}(\mathit{root}) \quad \triangleq \quad \mathit{GenerateComposite}(\textit{mk-Block}(\{\,\}, \mathit{root}))$$

When generating a composite model, a submodel is generated for each of its children. Where values are passed between submodels, these connections are also generated:

$$\mathit{GenerateComposite} : \mathit{Block} \to \mathit{CompositeModel}$$

$$\begin{array}{l}\mathit{GenerateComposite}(\mathit{node}) \quad \triangleq \quad \textit{mk-CompositeModel}(\\ \quad \{\mathit{id} \mapsto \mathit{GenerateSubmodel}(\mathit{node.contents.children}(\mathit{id}), \mathit{node.contents}, \mathit{id}) \mid \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{id} \in \mathbf{dom}\, \mathit{node.contents.children}\}, \\ \quad \{\mathit{GenerateConnection}(\mathit{flow}, \mathit{node}) \mid \\ \qquad\qquad \mathit{flow} \in \mathit{DetermineCTConnections}(\mathit{node.contents.connections})\})\end{array}$$

When building a submodel, the model name is assigned, and then either a composite or elementary submodel is created as appropriate. For each submodel, an interface is created in order to facilitate communication with other submodels:

$$\mathit{GenerateSubmodel} : \mathit{Block} \times \mathit{CompositeContent} \times \mathit{Id} \to \mathit{Submodel}$$

$$\begin{array}{l}\mathit{GenerateSubmodel}(\mathit{node}, \mathit{env}, \mathit{ref}) \quad \triangleq \quad \textit{mk-Submodel}(\\ \quad \mathbf{if}\ \textit{is-CompositeContent}(\mathit{node.contents}) \\ \quad \mathbf{then}\ \mathit{GenerateComposite}(\mathit{node}) \\ \quad \mathbf{else}\ \mathit{GenerateElementary}(\mathit{node}), \\ \quad \{\mathit{id} \mapsto \mathit{GeneratePort}(\mathit{node.ports}(\mathit{id}), \mathit{env}, \textit{mk-PortReference}(\mathit{ref}, \mathit{id})) \mid \\ \qquad\qquad\qquad\qquad\qquad \mathit{id} \in \mathit{DetermineCTPorts}(\mathit{node.ports})\})\end{array}$$

Each submodel interface includes ports facilitating connections between physical elements, or connections between a physical element and a controller. Connections between two controllers are not modelled in 20-sim. Ports to be added to 20-sim are calculated by evaluating the properties of the interface exposed by the port; ports are added if they expose an interface describing a physical transfer (identified by a flow port in SysML), or an interface bridging physical and control elements (identified by a service port in SysML, where the interface offered is a control interface, rather than a communication interface):

```
$DetermineCTPorts : Id \xrightarrow{m} PortType \to Id\text{-}\mathbf{set}$

$DetermineCTPorts(ports) \quad \triangleq$
    $\{id \mid id \in \mathbf{dom}\, ports \cdot (is\text{-}FlowPort(ports(id)) \lor$
                          $(is\text{-}ServicePort(ports(id)) \land is\text{-}ControlInterface(ports(id).exposes)))\}$
```

The orientation of each port is identified, before a suitable port implementation in 20-sim is calculated. In 20-sim, signal ports are used to facilitate communication between control and plant elements, and for facilitating physical transfers between components where the connection is not closed. Where Physical components are linked via a closed connection, bond ports are used instead:

```
$GeneratePort : PortType \times CompositeContent \times PortReference \to SubmodelPort$

$GeneratePort(port, env, ref) \quad \triangleq \quad mk\text{-}SubmodelPort($
    $DetermineOrientation(p, env, ref),$
    $\mathbf{if}\ (is\text{-}ServicePort(port) \lor (is\text{-}FlowPort(port) \land port.direction \neq \text{BOTH}))$
    $\mathbf{then}\ \text{SIGNAL}$
    $\mathbf{else}\ \text{BOND})$
```

Depending on the type of port, the appropriate orientation for the 20-sim representation of a port can be calculated either by evaluating properties of the port itself, or examining any connections utilising the port. In SysML, there are two types of port to consider: flow ports and service ports. In the case of the former, a direction is assigned to the port to indicate the direction in which material is permitted to pass through the port. Where a flow port is assigned a direction of simply ‘in’ or ‘out’, the 20-sim port can simply be assigned an orientation of ‘input’ or ‘output’ respectively. For flow ports with a direction of ‘both’, some evaluation of the use of the port is necessary. This is also the case for signal ports:

```
$DetermineOrientation : PortType \times CompositeContent \times PortReference \to Orientation$

$DetermineOrientation(port, env, ref) \quad \triangleq$
    $\mathbf{if}\ is\text{-}FlowPort(port)$
    $\mathbf{then\ cases}\ port.direction\ \mathbf{of}$
            $\text{IN} \to \text{INPUT}$
           $\text{OUT} \to \text{OUTPUT}$
          $\text{BOTH} \to DetermineFlowDirection(port, env, ref)$
          $\mathbf{end}$
    $\mathbf{else}\ DetermineInterfaceOrienation(port, env, ref)$
```

Whilst bidirectional flow ports are useful for indicating that a port might be modelled in 20-sim using bond notation rather than signal notation, this syntax does remove the ability to deduce an appropriate orientation for the port based on its properties alone. For ports of this type, item flows connect the port to other ports of a similar type. Since item flows are directional, it can be deduced that the where a port is used as the source to an item flow, the port should be assigned an orientation of ‘output’ in 20-sim. Similarly, where a port is used as the target to an item flow, the port should be assigned an orientation of ‘input’ in 20-sim:

```
$DetermineFlowDirection : PortType \times CompositeContent \times PortReference \to Orientation$

$DetermineFlowDirection(port, env, ref) \quad \triangleq$
    **let** $itemFlows = \{flow \mid flow \in env.connections \cdot is\text{-}ItemFlow(flow)\}$ **in**
    **if** $\exists flow \in itemFlows \cdot flow.source = ref$
    **then** OUTPUT
    **else if** $\exists flow \in itemFlows \cdot flow.target = ref$
        **then** INPUT
        **else** ...
```

This enables calculation of a 20-sim orientation for a flow port regardless of its assigned direction in SysML. This approach is limited however, and can only determine the orientation of a port if it is connected to some external element, as illustrated in Figure 5.3.

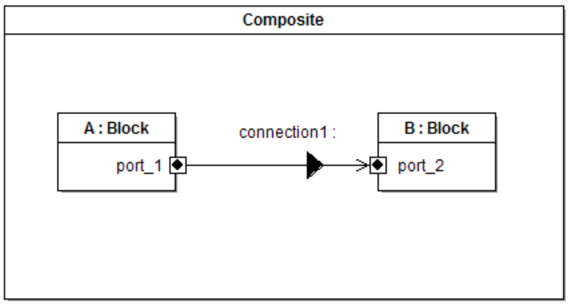


Figure 5.3: Externally connected bidirectional port example

It is possible that a port is used to offer some interface but it is not connected to another block. If the port is part of an interface for some elementary submodel, then a default orientation must be used. However, if the port is not connected externally but is part of a composite model, it may be the case that the port is used in a connection between the port and some sub-element of its owning element, as illustrated in Figure 5.4.

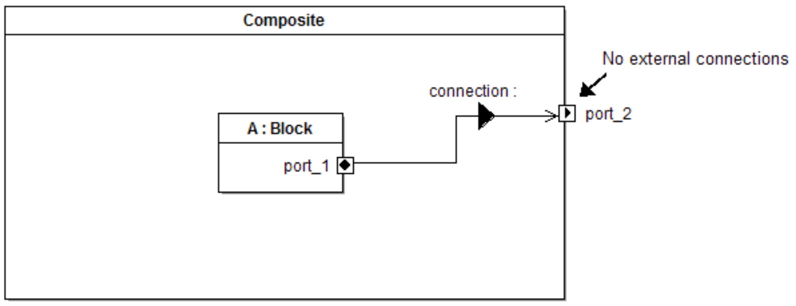


Figure 5.4: Parent port referencing example

In this case, port orientation can be determined by examining the internal connection. If the externally facing port is used as the source to an internal item flow, the port should be assigned an orientation of ‘input’ in 20-sim. Similarly, where an externally facing port is used as the target to an internal item flow, the port should be assigned an orientation of ‘output’ in 20-sim. If the port is not connected to anything at all, again a default orientation must be assumed:

```
$DetermineFlowDirection : PortType \times CompositeContent \times PortReference \to Orientation$

$DetermineFlowDirection(port, env, ref) \quad \triangleq$
    let $itemFlows = \{flow \mid flow \in env.connections \cdot is\text{-}ItemFlow(flow)\}$ in
    if $ref.owner \neq$ nil
    then if $\exists flow \in itemFlows \cdot flow.source = ref$
        then OUTPUT
        else if $\exists flow \in itemFlows \cdot flow.target = ref$
            then INPUT
            else if $is\text{-}CompositeContent(env.children(ref.owner).contents)$
                then $DetermineFlowDirection($
                    $p, env.children(ref.owner).contents,$
                    $mk\text{-}PortReference(\textbf{nil}, ref.port))$
                else INPUT
    else if $\exists flow \in itemFlows \cdot flow.source = ref$
        then INPUT
        else if $\exists flow \in itemFlows \cdot flow.target = ref$
            then OUTPUT
            else INPUT
```

A service port is used in SysML to expose some digital interface. Where this interface is utilised for communication between a control element and a plant element, the port must be added to 20-sim. Interfaces between controller and plant elements facilitate either reading some value from the environment (sense interface), or assigning some value to a plant component (actuate interface).

Unlike flow ports, service ports do not indicate a direction for permitted interaction, so the orientation of 20-sim representations of these ports must be calculated by examining any connections which utilise them. In contrast to item flows, information flows are not assigned a direction; instead, interfaces are either provided or required by ports. An information flow is always made between a port which provides an interface, and a port which requires it. In SysML, interfaces are always provided by the control element that defines them, with the associated plant element requiring the interface to either send or receive values to/from the controller.

Where a port provides an interface, it can be assumed to be assigning variables to/from a control element. If the interface offered is a sensing interface, then the port should become an ‘input’ port in order to facilitate passing data to the sensing controller. If the interface offered is an actuating interface, then the port should become an ‘output’ port in order to facilitating passing data from the actuating controller.

Where a port requires an interface, it can be assumed to be assigning variables to/from a plant element. If the interface offered is a actuating interface, then the port should become an ‘input’ port in order to facilitate passing data to the sensing controller. If the interface offered is an actuating interface, then the port should become an ‘output’ port in order to facilitating passing data from the actuating controller.

Figure 5.5 illustrates how the direction which data is passed can be calculated, making it possible to determine an orientation for the port.

As with bidirectional flow ports, if a service port is not connected to anything externally, but is part of an interface of a composite element, then the element can be inspected for internal connections utilising the port to determine its orientation.

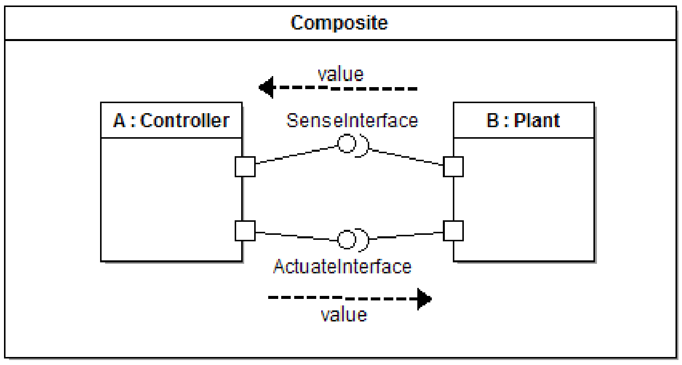


Figure 5.5: Passing values using information flows

```
$DetermineInterfaceOrienation : PortType \times CompositeContent \times PortReference$
                    $\rightarrow Orientation$

$DetermineInterfaceOrienation(port, env, ref) \quad \triangleq$
    **let** $informationFlows = \{flow \mid flow \in env.connections \cdot is\text{-}InformationFlow(flow)\}$ **in**
    **if** $ref.owner \neq$ **nil**
    **then if** $\exists flow \in informationFlows \cdot flow.providedBy = ref$
        **then if** $is\text{-}SenseInterface(port)$
            **then** INPUT
            **else** OUTPUT
        **else if** $\exists flow \in informationFlows \cdot flow.requiredBy = ref$
            **then if** $is\text{-}ActuateInterface(port)$
                **then** INPUT
                **else** OUTPUT
            **else if** $is\text{-}CompositeContent(env.children(ref.owner).contents)$
                **then** $DetermineInterfaceOrienation($
                    $p, env.children(ref.owner).contents,$
                    $mk\text{-}PortReference($**nil**$, ref.port))$
                **else** INPUT
    **else if** $\exists flow \in informationFlows \cdot flow.providedBy = ref$
        **then if** $is\text{-}SenseInterface(port)$
            **then** OUTPUT
            **else** INPUT
        **else if** $\exists flow \in informationFlows \cdot flow.requiredBy = ref$
            **then if** $is\text{-}ActuateInterface(port)$
                **then** OUTPUT
                **else** INPUT
            **else** INPUT
```

20-sim Connections are created to facilitate passing values between submodels. Submodels are connected to represent a physical transfer, or a control signal passed between adjacent ports. Connections to be added to 20-sim are calculated by evaluating the properties of the flow, and the properties of the ports connected by the flow; connections are added if they implement an interface describing a physical transfer (identified by an item flow in SysML), or an interface bridging physical and control elements (identified by an information flow in SysML, where the interface offered is a control interface, rather than a communication interface):

$$
\begin{aligned}
&\mathit{DetermineCTConnections} : \mathit{Flow}\text{-}\mathbf{set} \rightarrow \mathit{Flow}\text{-}\mathbf{set}\\
&\mathit{DetermineCTConnections}(\mathit{flows}) \quad \triangleq\\
&\qquad \{\mathit{flow} \mid \mathit{flow} \in \mathit{flows} \cdot (\mathit{is\text{-}ItemFlow}(\mathit{flow}) \lor\\
&\qquad\qquad\qquad (\mathit{is\text{-}InformationFlow}(\mathit{flow}) \land \mathit{is\text{-}ControlInterface}(\mathit{flow.implements})))\ \}
\end{aligned}
$$

For each connection created, a suitable implementation in 20-sim is calculated. If a connection is between signal ports, a signal is created, otherwise a bond port is created. Flows can be between ports owned by any two nested submodels, or between a nested submodel and its owning model. To facilitate this, the owning submodel is referenced so that its interface can be inspected:

$$
\begin{aligned}
&\mathit{GenerateConnection} : \mathit{Flow} \rightarrow \mathit{Connection}\\
&\mathit{GenerateConnection}(\mathit{flow}) \quad \triangleq\\
&\qquad \mathit{mk\text{-}Connection}(\mathit{flow.name}, \mathit{DetermineSource}(\mathit{flow}), \mathit{DetermineTarget}(\mathit{flow}))
\end{aligned}
$$

Continuous Time Equations

Each elementary submodel represents either some continuous time behaviour represented as a series of differential equations, or a link to a discrete event controller described in VDM-RT:

$$
\begin{aligned}
&\mathit{GenerateElementary} : \mathit{NodeDescription} \rightarrow \mathit{ElementaryModel}\\
&\mathit{GenerateElementary}(\mathit{node}) \quad \triangleq\\
&\qquad \mathbf{if}\ \mathit{is\text{-}PhysicalNode}(\mathit{node})\\
&\qquad \mathbf{then}\ \mathit{GeneratePhysical}(\mathit{node})\\
&\qquad \mathbf{else}\ \mathit{GenerateCyber}(\mathit{node})
\end{aligned}
$$

Continuous time behaviour is expressed via a seires of differential equations. For submodels of physical systems, these equations are constructed based on information encapsulated in SysML Parametric Diagrams. These equations may read or update a defined variable, or utilise some predefined value.

$$\mathit{GeneratePhysical} : \mathit{PhysicalNode} \to \mathit{ElementaryModel}$$

$$\begin{aligned}
&\mathit{GeneratePhysical}(\textit{mk-PhysicalNode}(\mathit{constraints}, \mathit{constants}, \mathit{bindings})) \quad \triangleq \\
&\quad \textit{mk-ElementaryModel}(\\
&\quad \{\mathit{id} \mapsto \mathit{GenerateParameter}(\mathit{constants}(\mathit{id})) \mid \mathit{id} \in \textbf{dom}\, \mathit{constants}\} \overset{m}{\cup} \\
&\quad \{b.\mathit{id} \mapsto \mathit{GenerateVariable}(b.\mathit{participants}, \mathit{constraints}) \mid b \in \mathit{bindings} \cdot \mathit{DetermineVariableBinding}(b)\}, \\
&\quad \{\mathit{BindVariables}(\mathit{constraint}.\mathit{body}) \mid \mathit{constraint} \in \textbf{rng}\, \mathit{constraints}\})
\end{aligned}$$

Any predefined parameters are assigned a valid type, and initialised to some value.

$$\mathit{GenerateParameter} : \mathit{Constant} \to \mathit{Parameter}$$

$$\mathit{GenerateParameter}(c) \quad \triangleq \quad \textit{mk-Parameter}(c.\mathit{value}, c.\mathit{unit})$$

Where a number of equations utilise or update a common value, a variable is added to the 20-sim submodel.

$$\mathit{DetermineVariableBinding} : \mathit{Binding}\textbf{-set} \to \mathbb{B}$$

$$\mathit{DetermineVariableBinding}(b) \quad \triangleq \quad \forall \mathit{ref} \in b.\mathit{participants} \cdot \textit{is-ParameterRef}(\mathit{ref})$$

$$\mathit{GenerateVariable} : \mathit{ParameterRef}\textbf{-set} \times \mathit{Id} \xrightarrow{m} \mathit{Constraint} \to \mathit{Variable}$$

$$\begin{aligned}
&\mathit{GenerateVariable}(\mathit{examples}, \mathit{env}) \quad \triangleq \quad \textbf{let } e \in \mathit{examples} \ \textbf{ in} \\
&\quad \textit{mk-Variable}(\mathit{env}(e.\mathit{constraint}).\mathit{parameters}(e.\mathit{parameter}))
\end{aligned}$$

Equations are specified is SysML using generic variable identifiers. Where the equations are used, these variables must be bound to the instance of the equation. This binding is not specified further.

$$\mathit{BindVariables} : \mathit{Equation} \to \mathit{Equation}$$

$$\mathit{BindVariables}(e) \quad \triangleq \quad \ldots$$

Where a submodel links the 20-sim model to a controller expressed in VDM-RT, values which are passed between models have local copies maintained. A local copy of each shared variable is added to a 20-sim placeholder submodel, and a series of equations map local copies to the co-simulation contract:

$$GenerateCyber : Block \times Id \to ControlLinkModel$$

$$\begin{aligned}
&GenerateCyber(node, id) \quad \triangleq \\
&\quad \textbf{let } ports = node.ports \textbf{ in} \\
&\quad \{GenerateExternalId(id, p) \mapsto DetermineExternalVariable(ports(p).exposes) \mid \\
&\quad p \in \textbf{dom}\, ports \cdot is\text{-}ControlInterface(ports(p).exposes)\}
\end{aligned}$$

Logical Composition

The structure of all elements is included in 20-sim, along with the equations outlining continuous time behaviours, and placeholders for digital elements. A discrete event model is then created in VDM-RT, and linked to the placeholders in 20-sim.

$$GenerateDEModel : CompositeContent \to DEModel$$

$$\begin{aligned}
&GenerateDEModel(root) \quad \triangleq \\
&\quad mk\text{-}DEModel(GenerateSystem(root), GenerateContract(root.children))
\end{aligned}$$

A co-simulation contract defines the interface between DE and CT models in terms of variables which are shared between the models. This contract is added to the DE model and specifies local identifiers of each shared variables in each model.

$$GenerateContract : Id \xrightarrow{m} Block \to Contract$$

$$\begin{aligned}
&GenerateContract(m) \quad \triangleq \quad DetermineSharedVariables(m) \overset{m}{\cup} \\
&\quad \overset{m}{\biguplus}(\{GenerateContract(m(c).contents.children) \mid c \in \textbf{dom}\, m \cdot is\text{-}CompositeContent(m(c).contents)\})
\end{aligned}$$

$$DetermineSharedVariables : Id \xrightarrow{m} Block \to Id \xrightarrow{m} SharedVariable$$

$$\begin{aligned}
&DetermineSharedVariables(m) \quad \triangleq \\
&\quad \overset{m}{\biguplus}(\{GenerateSharedVariables(node, m(node)) \mid node \in \textbf{dom}\, m \cdot DetermineCTLink(m(node))\})
\end{aligned}$$

$DetermineCTLink : Block \to \mathbb{B}$

$DetermineCTLink(node) \quad \triangleq \quad \exists p \in \textbf{rng}\, node.ports \cdot is\text{-}ControlInterface(p.exposes);$

$GenerateSharedVariables : Block \to Id \xrightarrow{m} SharedVariable$

$$\begin{aligned}
&GenerateSharedVariables(id, node) \quad \triangleq \\
&\quad \{CombineIds(id, p) \mapsto GenerateSharedVariable(node.ports(p).exposes) \mid \\
&\quad p \in \textbf{dom}\, node.ports \cdot is\text{-}ControlInterface(node.ports(p).exposes)\}
\end{aligned}$$

$GenerateSharedVariable : ControlInterface \to SharedVariable$

$$\begin{aligned}
GenerateSharedVariable(i) \quad \triangleq \quad & \textbf{if}\ is\text{-}SenseInterface(i) \\
& \textbf{then}\ \text{MONITORED} \\
& \textbf{else}\ \text{CONTROLLED}
\end{aligned}$$

A VDM-RT System class records each DE element to be modelled digitally. Controllers may be nested within composite systems, so all constituents and any child elements are examined for DE elements.

$GenerateSystem : CompositeContent \to System$

$$\begin{aligned}
&GenerateSystem(root) \quad \triangleq \quad \textbf{let}\ controllers = GenerateControllers(root.children)\ \textbf{in} \\
&\quad \{id \mapsto GenerateCPU(id, controllers(id)) \mid id \in \textbf{dom}\, controllers\}, \\
&\quad \{\{ResolveId(f.providedBy, root), ResolveId(f.requiredBy, root)\} \mid \\
&\quad f \in root.connections \cdot DetermineControlInterface(f)\}
\end{aligned}$$

$DetermineControlInterface : Flow \to \mathbb{B}$

$$\begin{aligned}
DetermineControlInterface(f) \quad \triangleq \quad & \textbf{if}\ is\text{-}InformationFlow(f) \\
& \textbf{then}\ is\text{-}ControlInterface(f) \\
& \textbf{else false}
\end{aligned}$$

A virtual CPU is created for each distinct DE element. This abstractly represents a unique computational entitiy for each DE constituent.

$$GenerateCPU : Id \times Block \to CPU$$

$$GenerateCPU(id, node) \quad \triangleq \quad \textbf{let } controller \in \{c \mid c \in node.contents \cdot is\text{-}ControllerClass(c)\} \textbf{ in}$$
$$\quad mk\text{-}VDMController(controller.className, DetermineControllerInterfaces(id, node))$$

With distinct digital elements added to a virtual execution environment, a mechanism for exchanging information between values must be established in order to facilitate communication between CPUs. Where any two CPUs share an interface, they are added to a virtual BUS.

$$DetermineControllerInterfaces : Id \times Block \to Id \xrightarrow{m} Id$$

$$DetermineControllerInterfaces(id, node) \quad \triangleq$$
$$\quad \{id \frown \_ \frown p \mapsto node.ports(p).exposes.interfaceName \mid$$
$$\quad p \in \textbf{dom}\, node.ports \cdot is\text{-}ControlInterface(node.ports(p).exposes)\}$$

$$GenerateControllers : Id \xrightarrow{m} Block \to Id \xrightarrow{m} Block$$

$$GenerateControllers(tree) \quad \triangleq \quad DetermineCyberNodes(tree) \overset{m}{\cup}$$
$$\quad \biguplus_{m} (\{GenerateControllers(tree(id).contents.children) \mid$$
$$\quad id \in \textbf{dom}\, tree \cdot is\text{-}CompositeContent(tree(id).contents)\})$$

$$DetermineCyberNodes : Id \xrightarrow{m} Block \to Id \xrightarrow{m} Block$$

$$DetermineCyberNodes(m) \quad \triangleq$$
$$\quad \{id \mapsto m(id) \mid id \in \textbf{dom}\, m \cdot is\text{-}CyberNode(m(id).contents)\}$$

```
$ResolveId : PortReference \times CompositeContent \to Id$

$ResolveId(p, env) \quad \triangleq$
    **let** $last = mk\text{-}PortReference(\textbf{nil}, p.port)$ **in**
    **if** $p.owner \neq \textbf{nil}$
    **then let** $node = env.children(p.owner).contents$ **in**
        **if** $is\text{-}CompositeContent(node)$
        **then** $ResolveId(last, node)$
        **else** $p.owner$
    **else let** $infoFlows = \{i \mid i \in env.connections \cdot is\text{-}InformationFlow(i)\}$ **in**
        **let** $flow \in infoFlows \cdot flow.providedBy = last \vee flow.requiredBy = last$ **in**
        **if** $flow.providedBy = last$
        **then** $ResolveId(last, env.children(flow.requiredBy.owner).contents)$
        **else** $ResolveId(last, env.children(flow.providedBy.owner).contents)$
```

5.3 Summary

This chapter describes an abstract representation of the SysML profile defined in Chapter 4 alongside a similarly abstract representation of co-models. VDM-SL is used to describe constructs of each language in terms of basic data types, and the VDM-SL descriptions enable the definition of a function to transform between constructs.

Whilst the transformation process does not translate between concrete models, the development of a mapping between formalisms provides confidence that a translation is viable, and is a precise specification for the implementation of a model transformation utility.

6 Approach Verification

To gain confidence in the proposed engineering approach, it must be verified. *Verification* describes the confirmation that a system or process satisfies specified requirements by the provision of objective evidence. Verification comprises a set of activities to ensure the correctness of a particular system or process, identifying any deviations from its specification (ISO/IEC/IEEE 15288 2015, p. 7).

A range of techniques might be employed to attempt verification of a system or process. Where properties or characteristics of a system of interest are best determined by observation, visual inspection might be employed to verify the system. Where properties or characteristics of a system can be quantitatively analysed, verification of a system might be achieved by testing, whereby a system or process is exposed to real conditions and its output measured. Where testing of real conditions is unattainable, verification of a system might be approached using mathematical or probabilistic analysis. Demonstration of system behaviour might also be employed in its verification, where correct operation is demonstrated against operational and observable characteristics without measurement. In verification by demonstration, a series of tests can illustrate that the response of the system to a given stimuli is suitable, where observations are compared to expected responses.

To verify the model-based engineering techniques described in chapters 4 and 5, the proposed approach can be demonstrated by its use in the description of an exemplary CPSoS. Demonstration of the approach can be facilitated by the application of the proposed techniques to a suitable case study. The use of an appropriate case study to demonstrate that the chosen system can be described using the proposed architectural framework together with the supporting SysML profile, can provide confidence in suitability of the framework. Demonstration that application of the co-model generation process to the resulting metamodel produces the expected co-model, can provide further confidence in the adequacy of the translation process. Co-simulation of the resulting co-model can provide additional confidence that the overall approach is satisfactory in addressing its stated criteria when applied to an exemplary system of interest.

This chapter describes the development of a CPSoS case study, including the selection of an appropriate study and the development of models to describe the selected system.

Section 6.1 describes the verification criteria against which models produced by the application of the approach should be compared. The criteria include requirements of the chosen study to ensure that the chosen system is an appropriate example of a CPSoS, as well as requirements of the proposed approach against which the success of the study can be assessed. Section 6.2 introduces the particular case to be studied. Both the system selected to demonstrate the approach, and the scenario to which the system will be exposed are described. Models of the chosen system are presented in Section 6.3, including an architectural description, a corresponding co-model, and results of its co-simulation. The co-model presented in this section is constructed manually, as a specification for the expected output from the co-model generation process. This process is verified in Section 6.4, which compares manual and automated translations of metamodels. Section 6.5 considers the strengths and weaknesses of the approach based on its demonstration by completion of the case study, including comparison of the proposed techniques with viable alternatives.

6.1 Verification Criteria

Verification of the proposed approach to model-based engineering of CPSoSs requires evaluation of the extent to which the proposed techniques satisfy the overall objectives of the approach. Section 6.1.1 decomposes the approach objectives to a set of requirements which provide a basis for the verification of the approach. Verification by demonstration requires selection of an appropriate case study. To constitute an appropriate example, the chosen case study should describe a suitable system of interest in the context of an appropriate scenario. The requirements of the chosen case are outlined in Section 6.1.2, including constraints over both the system and scenario.

6.1.1 Approach Requirements

In Section 1.4, a series of objectives are outlined which specify the particular challenges facing CPSoS engineering which the proposed approach to their model-based development should overcome. Each objective can be decomposed into a set of requirements, against which the approach itself can be verified.

Defining Architectural Descriptions

The proposed approach should facilitate the architectural description of a system. Facilitation of such descriptions is constrained by the following requirements:

- Terms and concepts important to the specification of a CPSoS must be defined, and the relationships between concepts identified. The ability to express a system in terms related to CPSoS engineering is essential in the construction of meaningful models.
- A collection of important considerations of a system of interest must be specified. Guidance must include constraints for ensuring that an architectural description is adequate in expressing the system with sufficient detail for supporting the production of executable models.

Realising Architectural Descriptions

The proposed approach should enable the realisation of architectural descriptions. Enabling the realisation of architectural descriptions is constrained by the following requirements:

- For each element specified in an architectural description, expression of the element must be possible using an architectural definition language. An appropriate language must be independent from any particular engineering domain, and must include mechanisms for the expression of both discrete and continuous phenomena.
- Guidelines for the use of an architectural definition language must explicitly relate architectural elements to language constructs.

Enabling Simulation

The proposed approach should support the production of executable models based on architectural descriptions. The production of executable models can be automated by the mapping of information between complementary formalisms. Supporting such a mapping is constrained by the following requirements:

- Executable models must utilise complementary formalisms to ensure high fidelity of modelled phenomena of both a continuous and discrete nature.

- Complementary abstractions of a system must be consistent. This demands that any properties or characteristics specified at an architectural level must be preserved in an executable equivalent.

6.1.2 Case Study Requirements

Demonstration of the proposed approach to model-based engineering of CPSoSs can be facilitated by the application of the proposed techniques to a suitable *case study*. A case study is an empirical enquiry which facilitates the exploration of a particular phenomenon (the *case*) in its environmental context to gain detailed insight into some property or behaviour (Ridder 2017, Yin 2014). A case might be an organisation or person, an event, or some problem or an anomaly (Burawoy 2009, Stake 2005, Yin 2014). In contrast to experimental methods, the contextual conditions of a case study are not controlled but are part of the investigation. Given this, it is important that the context is constructed to be sensitive to reality (Bonoma 1985), enabling an understanding of the dynamics of a chosen setting (Eisenhardt 1989). Exercising the proposed techniques by their application to an appropriate case study will highlight any benefits of the approach as well as challenges to its implementation (Ryan et al. 2002).

For a case study to be suitable for the verification of the proposed techniques described in chapters 4 and 5, a case must be chosen which satisfies a series of both technical and contextual requirements. When designing a scenario to be modelled, it is essential that these requirements are satisfied for the case study to be an appropriate mechanism for demonstrating the effectiveness of all aspects of the proposed approach.

Case Requirements

Firstly, the chosen case must present a system which exhibits characteristics typical of a SoS. In satisfying this requirement, the case study must describe a system composed of further constituent systems. These constituents should be independently owned and/or managed and might also be a composite of additional subsystems. Furthermore, there should be some level of coordination between constituents, with interaction across the system boundary.

The chosen case must also describe a system which exhibits characteristics typical of a CPS. In satisfying this requirement, the chosen system must include constituents exhibiting both discrete and continuous phenomena. Constituents must comprise a collection of elements which present significant challenges to be solved using a continuous time formalism, as well as significant challenges to be solved using discrete techniques. Furthermore, there should be some level of cooperation between cyber and physical components, with interaction across the continuous/discrete boundary.

Scenario Requirements

A case study presents a representation of a systems behaviour over some period of time. In selecting the series of events to be included in the study, a *scenario* should be developed which addresses real concerns and uses real information and data from the selected domain. A scenario is a description of what system stakeholders might do and/or experience as they make use of a particular system (Carroll 1995), and includes specification of the capabilities and starting state of constituent elements, description of any agents which might influence (or be influenced by) the scenario, and an outline of a particular sequence of actions (Potts 1995, Whitworth et al. 2006).

Scenarios are often open-ended and informally defined. They are focussed on enhancing communication of a particular sequence of events rather than concerned with consistency, rigour and completeness (Beynon-Davies & Holmes 2002). Scenarios are typically user-centric, and provide an intermediate abstraction between formal system specifications and informal discussions between system developers and users (Carroll 1995). Whilst a scenario is typically expressed as a narrative, it might also comprise storyboards, multi-media mock-ups, scripted prototypes, or physical situations arranged to support particular activities (Kuuti 1995).

For a scenario to provide a plausible approximation of outcomes of a series of events, it must be representative of a real set of circumstances. The development of a well-grounded scenario should use existing research and data about the scenario domain, and scenarios developed for similar investigations might be repurposed. The development of a plausible scenario might utilise publications and internal documents describing a system of interest, or benefit from consultation with domain experts (Leney et al. 2004).

6.2 Scenario Development

The rail sector is identified by Engell, Paulen, Sonntag, Thompson, Reniers, Klessova & Copigneaux (2015) as an important example of a CPSoS. Across Europe, existing railway infrastructure is controlled by a consortium of control centres, each responsible for route planning and incident management over a particular area. Operation of each control centre depends on the coordination of a large collection of constituent systems, which themselves are composed of a complex network of cyber and physical elements. Control centres are typically responsible for a particular geographical region, demanding close interaction between control centres of neighbouring regions.

Increasing costs of congestion, and a demand for reduction in transport emissions has led to the introduction of transport regulations in Europe which require the majority of medium- and long-distance journeys in the future to be made by rail (European Commission 2011). Centralisation of rail traffic management by the introduction of the European Railway Traffic Management System (ERTMS)[1] is essential in overcoming operational challenges of running trains between countries, but to become the preferred mode of transport, the rail industry must develop and deploy a range of new technologies to increase both capacity and resilience and reduce costs.

Given the partial autonomy of subsections of rail network as well as both private and public service providers using shared infrastructure, there is confidence throughout the rail industry that an SoS approach is essential in engineering new technologies. The adoption of an SoS approach is expected to contribute to increased network capacity by schedule optimisation, and reduced emissions by optimisation of driving strategies. Modelling and simulation are essential in both optimisation of complex systems and in motivating key stakeholders to migrate towards an SoS approach. High fidelity models are important to facilitate trustworthy simulation, and models must support detailed representation of both cyber and physical phenomena (Engell, Paulen, Sonntag, Thompson, Reniers, Klessova & Copigneaux 2015).

It is important that rail systems are optimised to increase capacity and decrease emissions. However, it is vital that any effort towards this does not compromise strict safety standards[2]. Many systems contribute towards the safe movement of trains through a network, where safe control of train movements is enforced by interlocking of resources to prevent conflicting movements.

Section 6.2.1 introduces concepts related to railway interlocking, including an overview of rail infrastructure and rolling stock, the coordination of trackside apparatus to control movement of rolling stock, and the principles which underpin effective interlocking. Section 6.2.2 outlines the constituent systems to be included in an interlocking scenario, and Section 6.2.3 describes the particular sequence of events which adequately demonstrate system operation.

[1]See `http://www.ertms.net/`
[2]See `http://www.rssb.co.uk/`

6.2.1 Rail Interlocking

A railway system comprises both physical and logical elements. Physical equipment includes static infrastructure such as trackwork, signalling equipment, stations, and power supply lines, as well as rolling stock such as cars and locomotives. Further to this, a system of operating rules and procedures for both safe and efficient operation is an essential component of a railway system (Pachl 2009).

Railway Infrastructure

Core to all railway infrastructures are *tracks*. A track comprises a pair of parallel rails fastened to series of ties. Tangential branching of tracks is facilitated by *turnouts*, an assembly of rails and movable points which enable rolling stock to run over one of two tracks. Turnouts can be arranged into some common track arrangements, including *junctions*, where two tracks converge to form a single track, and *crossovers*, which provide a connection between two parallel tracks. More complex arrangements such as *ladders* and *trees* combine multiple junctions to provide access to any of several parallel tracks.

Tracks are often complemented by signalling apparatus for coordinating train movements. *Fixed signals* indicate if a train can progress past the signal position, and can be controlled or automatic. Controlled signals are locally or remotely controlled by an operator, whilst automatic signals are affected by trains moving along a track. Signal indications can be actuated by semaphore or light signals, or a combination of both, and signals can be preceded by a number of distant signals, providing early indication of the signal status at the main signal position (RSSB 2014).

Tracks may also include equipment for the detection of train movements. *Train detection systems* typically provide input to signalling control systems, indicating the presence or absence of vehicles in a designated span of track, or if a train is passing or has passed a specific position (RSSB 2000). Types of train detection systems include *track circuits* and *axle counters*. Track circuits are electrical circuits which pass a low current through sections of track rails delimited by insulated rail joints. When the track section is occupied by a train, the axels produce a short circuit across the two rails, activating a relay. Axel counters include a counting head which increments a counter when passed by an axel. Counting heads often include a pair of sensors to enable detection of the direction of movement and speed of a passing train.

Rolling Stock

The primary vehicles in railway systems are *trains*. Trains describe *locomotives*, either alone or coupled to one or more vehicles, with authority to occupy track under operating conditions specified in a timetable. Locomotives are self-propelled railway vehicles that provide motive power for a train. Trains move along tracks using *bogies*, chassis carrying flanged wheels on one or more axels. Two bogies are typically fitted to each locomotive or carriage, each able to rotate independently to enable the vehicle to manoeuvre around curved track sections.

The tractive effort exerted by a locomotive is limited by the maximum force that can be produced by the driving machine, and the maximum force that can be transmitted by adhesion between the locomotive wheels and the track rail. Opposing the tractive effort of a locomotive are a variety of resistances, introduced by both track and train. Track resistance includes grade resistance and curve resistance, whilst train resistance is a composite of rolling resistance, bearing resistance, wind resistance, and additional factors such as dynamic processes (vibrations) and resistance of the traction system.

Signalling and Control

For ensuring safe separation of trains, tracks are often divided into *block sections*, which may be exclusively occupied by a single train. A train must not enter a block section if it is occupied by another train (RSSB 2014). Traditionally, tracks have been divided into *fixed blocks*, although technological advances have enabled implementation of *moving blocks* on some modern railways.

In a fixed block railway, track is divided into stationary block sections, typically delimited by block signals which indicate authority to move between blocks. Additionally, block movement authority might be communicated to the driver via cab signal systems, particularly on high speed railways where lineside signalling cannot be relied on. To authorise a signal for a train to enter a block section, the train ahead must have cleared the block section (and any dedicated overrun distance, if applicable).

In a moving block railway, spacing of trains is determined by a calculated braking distance. A following train must maintain a minimum distance from a leading train equal to the sum of a calculated braking distance of the rear train and an agreed safety distance. Whilst moving block separation can increase capacity of a railway in some cases, its implementation is hindered by the availability of prerequisite technology. Moving block railways require sophisticated techniques for the accurate detection of train ends, though advances in radio-based operating technologies is making the approach more feasible in modern applications.

Movement authority to enter a block is granted using *block signals*, which display information about the occupation of an upcoming block section. Block signals might be coupled with additional signalling equipment or markers to relay information about speed restrictions or the particular route over which the train is being sent. Block signals display information according to the principles of either *single-block signalling* or *multiple-block signalling*.

In a single-block signalling system, the indication of a block signal depends only on the state of the block section immediately beyond the signal. Signals can display a *danger* aspect to indicate that a train must come to a stop before the next block section, or a *clear* aspect to indicate authority to proceed to the next block. Each block requires a distant signal placed at an appropriate distance to the block signal to facilitate sufficient time for an approaching train to come to a complete stop before the end of the block.

In a multiple-block signalling system, the indication of a block signal depends on the state of two or more block sections beyond the signal. Alongside clear and danger aspects, signals might display an additional *caution* aspect to indicate that a train is clear to proceed, but should be prepared to stop at the next signal. This early indication of upcoming danger eliminates the requirement for a distant signal in each block. In three-block signalling, a further *preliminary caution* aspect can give advance warning of an upcoming caution aspect, often used in high speed railways where early notification of upcoming danger is essential to facilitate sufficient stopping distance.

Interlocking Principles

In railways, *interlocking* typically describes both the trackwork and signalling infrastructure that prevents conflicting movements through an arrangement such as junctions or crossings, and the control principles to achieve safe coordination of trackside equipment. Interlocked turnouts and signals are interconnected in such a way as to ensure that movements are coordinated in a safe sequence, and are controlled by either a local interlocking station or from a remote control centre (RSSB 2003).

Signals that govern train movements through an interlocking are described as *interlocking signals*, and interlocking signals governing entrance to an interlocking are described as *home signals*. Where home signals are followed by consecutive interlocking signals, intermediate blocks are used to enforce train separation within the interlocking. Interlocking signals governing train movements to leave an interlocking are described as *exit signals*. In interlockings with home signals but no consecutive interlocking signals, home signals grant authority to run through the entire interlocking into the next block section. In this case, home signals are also block signals, and opposing home signals double as exit signals.

Establishing a safe route through an interlocking requires a number of considerations. Firstly, all turnouts must be set and locked prior to a home signal being cleared for an approaching train. Turnout points and are considered interlocked with a signal when a signal is prevented from showing a clear aspect until the points have reached their intended position, and the points are prevented from being moved for as long as the signal aspect remains clear. Once an oncoming train has passed a clear signal, any interlocked points must remain locked until they have also been cleared. This locking is independent to the signal interlocking, and the combination of both locking functions is described as *route locking*.

Interlocking of points and signals prevents conflicting routes which require the setting of turnout points in opposing directions. Where conflicting routes through an interlocking do not require opposing point settings, additional interlocking mechanisms are required to prevent authorisation of conflicting movements. Specifically, locked routes must also lock any signals which might authorise a conflicting movement. Prevention of trains entering a locked route is described as *flank protection*, and can be implemented using operating rules or by additional trackside infrastructure such as *stop signals* and *derails*.

6.2.2 Constituent Systems

The development of models to support a rail interlocking case study must include representation of static infrastructure including track and signalling equipment, as well as the dynamics of any rolling stock.

Track Infrastructure

It is important that a model of railway infrastructure represents a plausible arrangement of tracks and signalling equipment. Ensuring this can be facilitated by the use of data describing real railway systems, however details of track layouts and supporting signalling equipment placement and capabilities is often commercially and/or security sensitive, and not available in the public domain. To overcome this, synthesised layouts can be used to illustrate typical rail scenarios. Synthesised infrastructures should be based on real systems, but can be obfuscated in such a way to be suitably anonymous for publication.

Synthesised layouts of rail infrastructure for reasoning about railway safety and capacity are included in the SafeCap platform (Iliasov et al. 2013). An output of the SafeCap project[3], the SafeCap platform uses a domain specific language to support the design of rail junctions by the provision of safety checking mechanisms and capacity evaluation tools. The platform includes a library of publicly accessible track layouts, based on data collected from real examples in the UK. These examples can be employed in the development of an interlocking case study.

Interlocking is required when a particular layout of track can permit conflicting movements if not properly managed. One common example of this is facilitation of bidirectional traffic over a single span of track. Whilst a pair of parallel tracks are often used to service concurrent bidirectional traffic along a particular route, this isnt always possible. An example from the SafeCap library describes a narrow bridge which is only able to facilitate a single span of track. With potential for bidirectional traffic from multiple branches either side of the bridge, the central section must be appropriately interlocked before train movements to cross the bridge can be authorised.

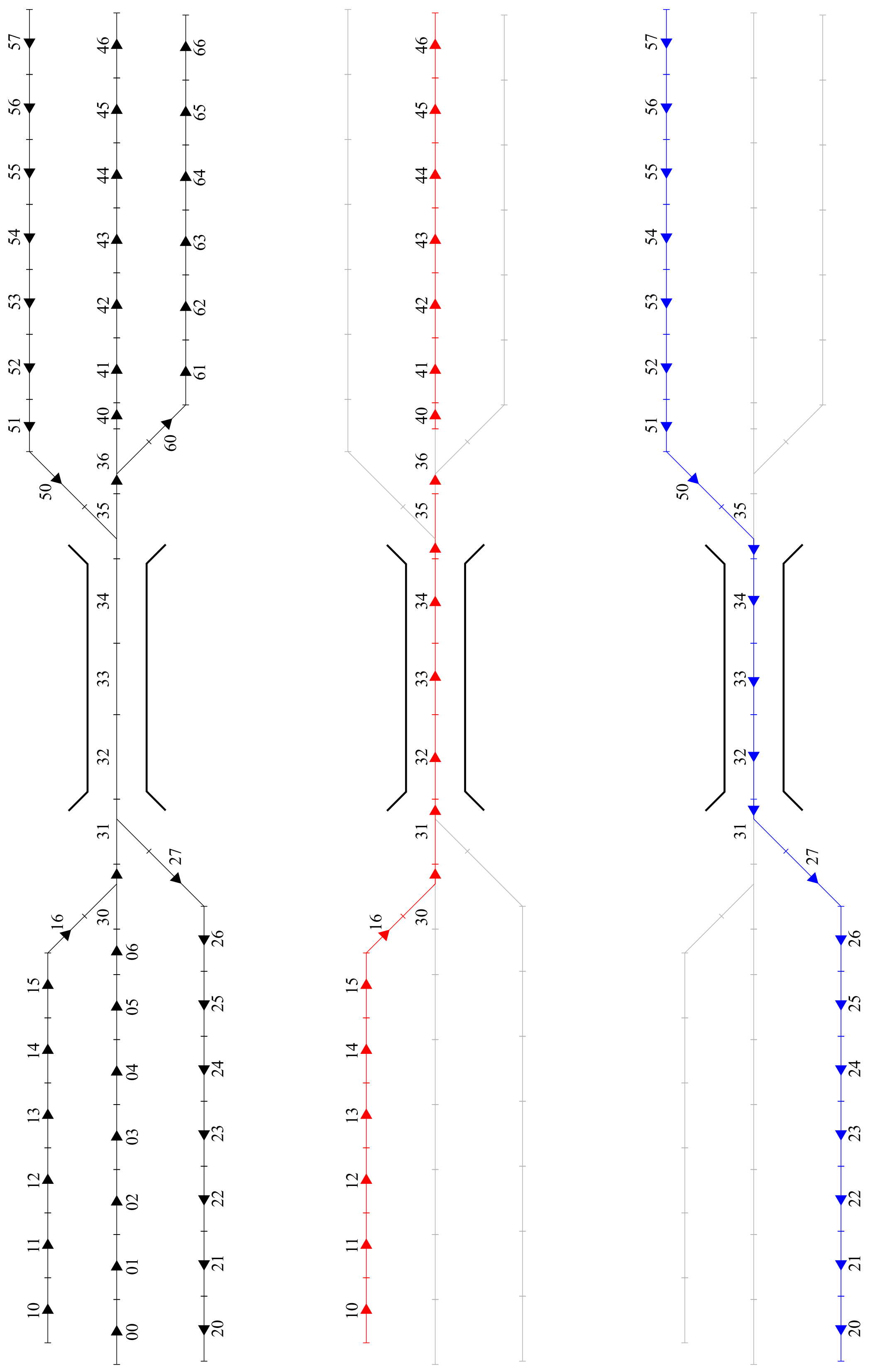
The narrow bridge example is illustrated in Figure 6.1a, which shows the division of track into fixed blocks. Each block is labelled with an integer identifier, and where a block is dedicated to unidirectional traffic, an arrow indicates the direction of travel. The length of each block as well as a maximum permitted speed is summarised in Table 6.1.

Models need not include a detailed representation of the layout of signalling apparatus. For modelling the control of trains, an abstract representation of signalling infrastructure must provide continuous indication movement authority to progress past a particular block section. Whilst implementation of signalling might employ block signals combined with distant signals or radio-based in-cab signalling, this detail is not important to the scenario.

Similarly, infrastructure models should include an abstract representation of train detection mechanisms. The occupied status of each block section should be detectable, though the implementation of this (e.g. track circuits, axel counters) is not important to the scenario.

Models must include a representation of turnouts which enables trains to pass over any valid routes. Whilst operation of turnouts is not instantaneous, it is sufficiently trivial not to be modelled explicitally.

[3]See `http://safecap.co.uk/`



(a) Track topology (b) $route_{r1}$ (c) $route_{r2}$

Figure 6.1: Track topology and chosen routes

Table 6.1: Track Section Properties

Segment	**Length** (m)	**Speed** (m/s)
00	500	17.8
01	500	17.8
02	500	17.8
03	500	17.8
04	500	17.8
05	500	17.8
06	350	17.8
10	500	17.8
11	500	17.8
12	500	17.8
13	500	17.8
14	500	17.8
15	500	17.8
16	400	17.8
20	500	17.8
21	500	17.8
22	500	17.8
23	500	17.8
24	500	17.8
25	500	17.8
26	500	17.8
27	600	17.8
30	500	8.9
31	500	8.9
32	650	8.9
33	600	8.9
34	650	8.9
35	500	8.9
36	500	20
40	200	17.8
41	500	17.8
42	500	17.8
43	500	17.8
44	500	17.8
45	500	17.8
46	500	17.8
50	600	17.8
51	400	17.8
52	500	17.8
53	500	17.8
54	500	17.8
55	500	17.8
56	500	17.8
57	500	17.8
60	400	17.8
61	500	17.8
62	500	17.8
63	500	17.8
64	500	17.8
65	500	17.8
66	500	17.8

Rolling Stock

Models approximating the dynamics of train movement should be reflective of real trains in operation on modern lines. Train movement depends on fixed parameters (e.g. mass) as well as variables (e.g. maximum tractive force) which change depending on factors such as velocity. Modelling dynamic properties mathematically is a highly complex problem, so such values are typically approximated based on measured performance in calibration testing (Pachl 2009).

An example of dynamic properties determined by empirical experimentation are resistances opposing locomotion of a train. Train resistance is typically approximated as a function of a train's speed by the *Davis Equation*, a quadratic function over values obtained by fitting equation coefficients to data from run down tests (Rochard & Schmid 2000). These coefficients include the calculated coefficient of bearing resistance (a), flange resistance (b) and aerodynamic resistance (c).

Alongside data describing plausible rail infrastructures, the SafeCap library also includes reference data for a range of trains. One such example is the InterCity 125, a high speed train in regular service in the UK. Important data for modelling the dynamics of the InterCity 125 is summarised in Table 6.2.

Table 6.2: InterCity 125 Constants

Property	**Value**
Mass (kg)	446000
Total length (m)	220
Maximum braking force (N)	147247
Bearing resistance coefficient	3222
Flange resistance coefficient	112.8
Air resistance coefficient	7.802

The maximum tractive effort which can be exerted by a locomotive is constrained by the speed at which the train is moving. The maximum tractive effort measured at a range of running speeds is summarised in Table 6.3. Maximum tractive effort for speeds other than those explicitly sampled can be approximated by interpolating between the sampled values. A higher fidelity approximation can be calculated using regression analysis. Regression analysis of the sampled data enables the expression of the maximum tractive effort for the InterCity 125 as a function of its speed, as:

$$f(x) = 0.0469x^4 - 8.8717x^3 + 641.05x^2 - 22507x + 393805$$

The strength of this relationship is illustrated by plotting the calculated function against the original sampled data in Figure 6.2. The data indicates that maximum tractive effort is limited at $160000N$ ($MaxForce$) regardless of speed, so the function is limited to enforce this. Furthermore, actual tractive effort is a factor of the maximum tractive effort, determined by throttle input. Including both additional considerations, the tractive effort can be expressed as a function of both speed (s) and throttle (t), as:

$$thrust = limit(0.0469s^4 - 8.8717s^3 + 641.05s^2 - 22507s + 393805, 0, MaxForce) \cdot t$$

Table 6.3: InterCity 125 Tractive Effort Measurements

Speed (m/s)	**Maximum Tractive Effort** (N)
00.0	160000
16.6	160000
19.4	140000
21.7	127000
24.4	114000
27.6	102000
31.9	88600
37.7	75900
44.6	63500
52.8	51600
55.9	47100

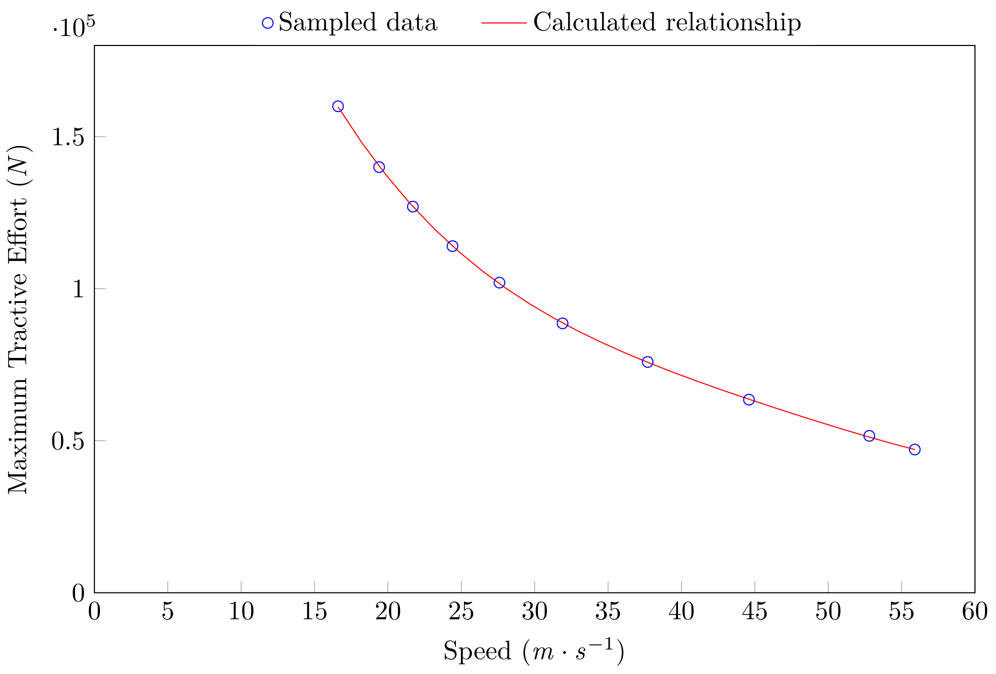


Figure 6.2: InterCity 125 Maximum Tractive Effort

The energy consumption of a locomotive is dependent on both its running speed and exerted tractive effort. The power draw measured at a range of running speeds whilst exerting a range of efforts is summarised in Table 6.4.

Plotting these values in Figure 6.3 illustrates a linear relationship between power draw and speed for a given tractive effort, and that power draw at a given speed is proportional with varying tractive effort. This proportionality makes it appropriate to interpolate between measurements for minimum and maximum tractive effort measurements.

A minimum tractive effort ($0N$) gives a constant power draw. This *ambient energy use* of $174000Js^{-1}$ is described as $MinPower$, and non-locomotive systems such as cabin lighting and air conditioning contribute significantly to this energy use when the driving engine is idle. Regression analysis of power draw measurements at a maximum possible tractive effort enables the expression for energy demand at a maximum possible tractive effort for the InterCity 125 as function of its speed, as:

$$f(x) = 166855x + 567045$$

Using both known values for maximum possible tractive effort ($MaxForce$) and minimum possible power draw ($MinPower$), power draw at a given speed can be calculated for any tractive effort by interpolation:

$$power = (166855 \cdot speed + 567045 - MinPower) \cdot \frac{force}{MaxForce} + MinPower$$

The accuracy of this approximation can be illustrated by plotting calculated values for each of the sampled data points against the original sampled data in Figure 6.4.

6.2.3 Event Sequence

The model can be used to engineer operating procedures, where simulation of the model can assess effectiveness of the solution for ensuring safety. Further to a provision of confidence in the safety of the implemented procedures, the model can be used to evaluate the effectiveness of the proposed techniques in increasing capacity by optimising throughput, and reducing emissions by optimising driving strategies.

Table 6.4: InterCity 125 Power Consumption (W) Measurements

Speed	**Tractive Effort** (N)				
(m/s)	**0**	**40000**	**80000**	**120000**	**160000**
0	174000	273000	372000	471000	570000
16.6	174000	963142	1752283	2541425	3330566
19.4	174000	1078023	1982045	2886068	3790090
21.7	174000	1181772	2189543	3197315	4205087
24.4	174000	1303749	2433499	3563248	—
27.6	174000	1436897	2699794	3962691	—
31.9	174000	1634447	3094895	4555342	—
37.7	174000	1893442	3612885	—	—
44.6	174000	2210966	4247931	—	—
52.8	174000	2604273	5034546	—	—
44.6	174000	2740339	5306678	—	—

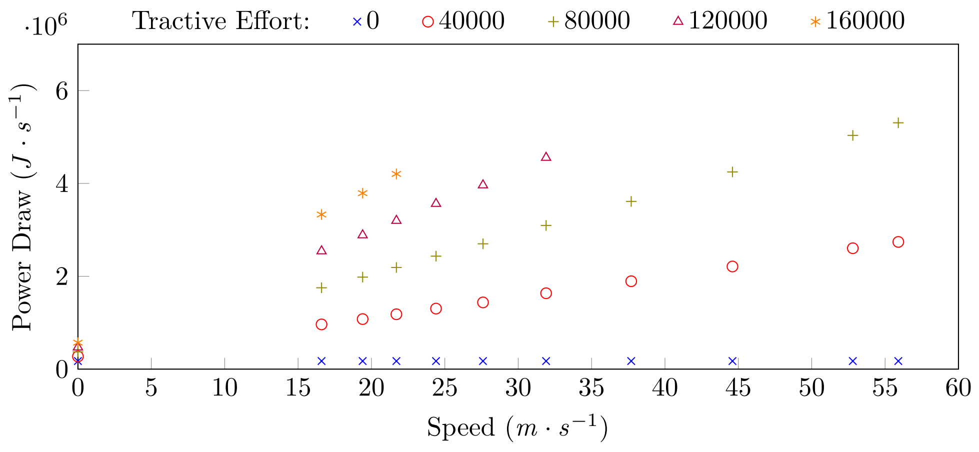


Figure 6.3: InterCity 125 Power Draw Measurements

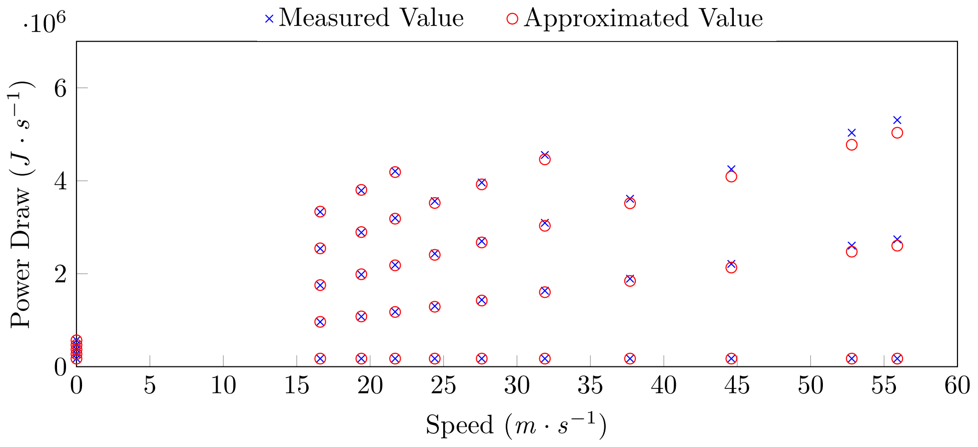


Figure 6.4: InterCity 125 Power Draw Approximations

To demonstrate the impact of interlocking procedures over the specified infrastructure, two conflicting routes should be scheduled. A scenario should include two trains attempting opposing movements over a common resource. In the example of the narrow bridge infrastructure outlined in Figure 6.1a, an illustrative example of conflicting movements includes the simultaneous approach of two trains travelling in opposing directions.

Scheduling of routes can be represented as a sequence of track blocks, where the movement of a train progresses over the track in order of sequence. The progression of a train over its scheduled route is subject to movement authority to enter each block, granted by signalling equipment along each route. Two conflicting routes—$route_{r1}$ and $route_{r2}$—are defined as:

$$\begin{aligned} route_{t1} = \{ & \ 10 \mapsto 11 \mapsto 12 \mapsto 13 \mapsto 14 \mapsto 15 \mapsto 16 \mapsto 30 \mapsto 31 \mapsto 32 \mapsto 33 \mapsto 34 \mapsto 35 \mapsto \\ & 36 \mapsto 40 \mapsto 41 \mapsto 42 \mapsto 43 \mapsto 44 \mapsto 45 \mapsto 46 \ \} \end{aligned}$$

$$\begin{aligned} route_{t2} = \{ & \ 57 \mapsto 56 \mapsto 55 \mapsto 54 \mapsto 53 \mapsto 52 \mapsto 51 \mapsto 50 \mapsto 35 \mapsto 34 \mapsto 33 \mapsto 32 \mapsto 31 \mapsto \\ & 27 \mapsto 26 \mapsto 25 \mapsto 24 \mapsto 23 \mapsto 22 \mapsto 21 \mapsto 20 \ \} \end{aligned}$$

The two routes are further illustrated in figures 6.1b and 6.1b, respectively.

6.3 Model Description

Verification of the proposed approach to model-based engineering of CPSoSs requires the construction of both an architectural model and a corresponding co-model. Manual construction of an architectural description can verify the suitability of the architectural framework and supporting SysML profile for the expression of important characteristics of the system of interest described in Section 6.2. The specification of an architectural description is outlined in Section 6.3.1.

Manual construction of a corresponding co-model can be used as a specification for the verification of the co-model generation process. The specification of an appropriate co-model is described in Section 6.3.2. Verification of the co-model is aided by its co-simulation, the results of which are described in Section 6.3.3.

6.3.1 Architectural Description

An architectural description of the system is constructed by employing the proposed SysML profile to realise each of the viewpoints of the architectural framework described in Chapter 4.

Composite Structures

The rail interlocking system to be described is composed of three constituents: the track infrastructure and two trains. To satisfy the requirements of the architectural framework, this composite system must be expressed in terms of its structure (by the construction of a Composite System Structure View (CSSV)) and any relationships between component parts (by the construction of a Composite System Connections View (CSCV)). SysML is used for the construction of each of these system views, in figures 6.5 and 6.6, respectively.

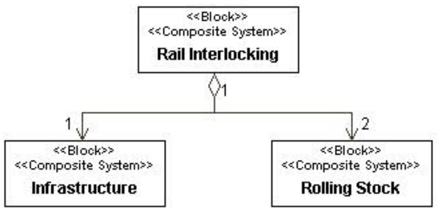


Figure 6.5: Composite System Structure View for Rail Interlocking CPSoS

Both the *Infrastructure* and *Rolling Stock* constituents introduced in the CSSV of the system are described as further composite systems. To satisfy the constraints of the architectural framework, a

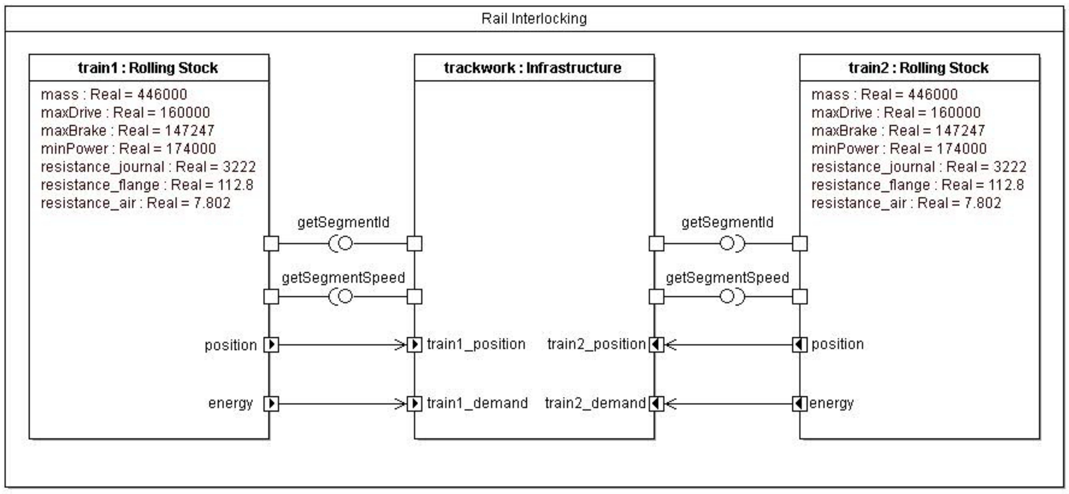
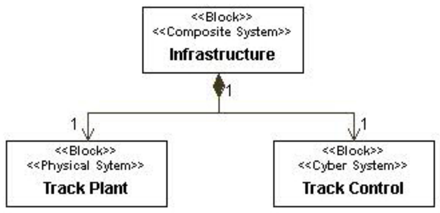
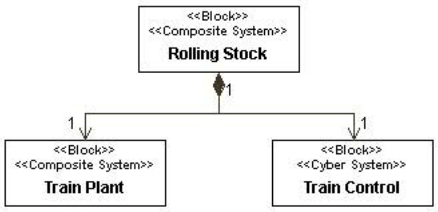


Figure 6.6: Composite System Connections View for Rail Interlocking CPSoS



(a) CSSV for Infrastructure



(b) CSSV for Rolling Stock

Figure 6.7: Composite System Structure Views for Rail Interlocking CPSoS constituents

CSSV and CSCV must also be specified for each of these elements. Internal structural elements are defined for each constituent in Figure 6.7, and connections between these elements are described in figures 6.8 and 6.9.

In Figure 6.8, the interface between *Track Plant* and *Track Control* has been cropped in an effort to aid clarity. Where the model includes an explicit interface between these two models for each of the modelled track sections, only a subset of these interfaces are included in the figure.

Whilst the majority of constituents of both *Infrastructure* and *Rolling Stock* are either cyber or physical, one additional composite system is introduced in *Train Plant*. A further CSSV and supporting CSCC must define the inner elements of the *Train Plant* constituent, described in figures 6.10 and 6.11, respectively.

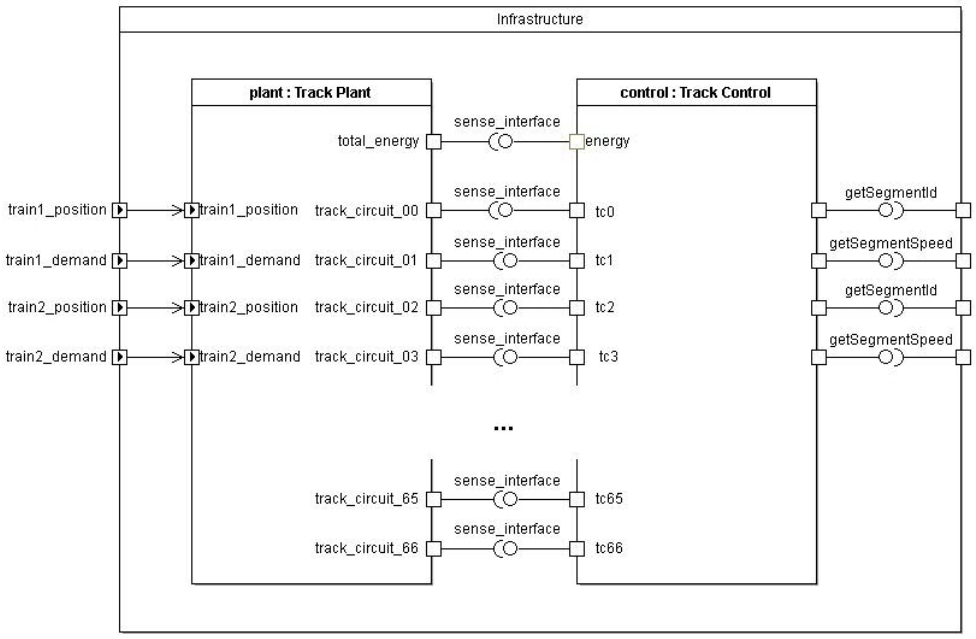


Figure 6.8: Composite System Connections View for Infrastructure constituent

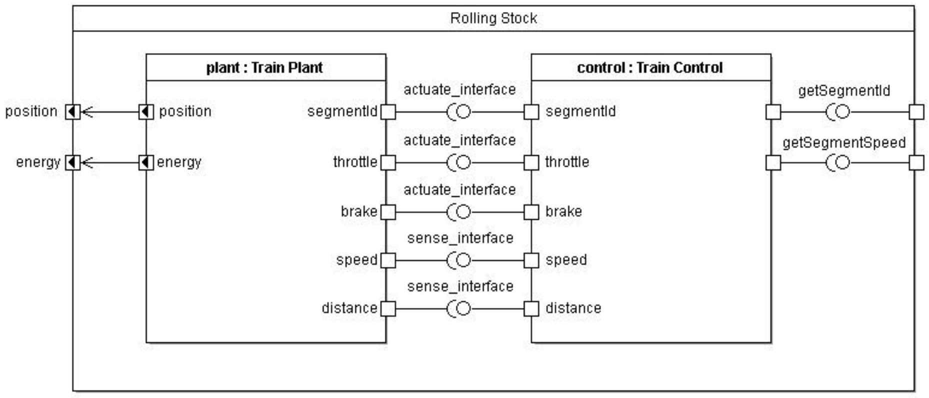


Figure 6.9: Composite System Connections View for Rolling Stock constituents

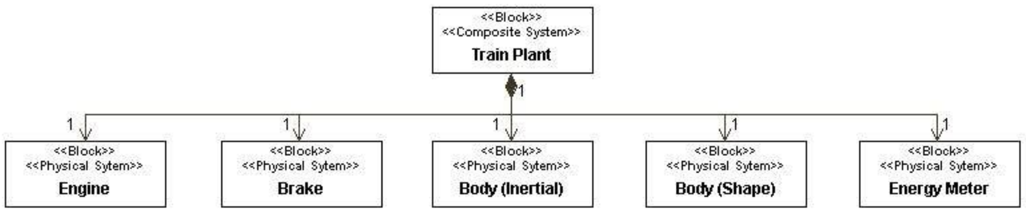


Figure 6.10: Composite System Structure View for Train Plant constituent

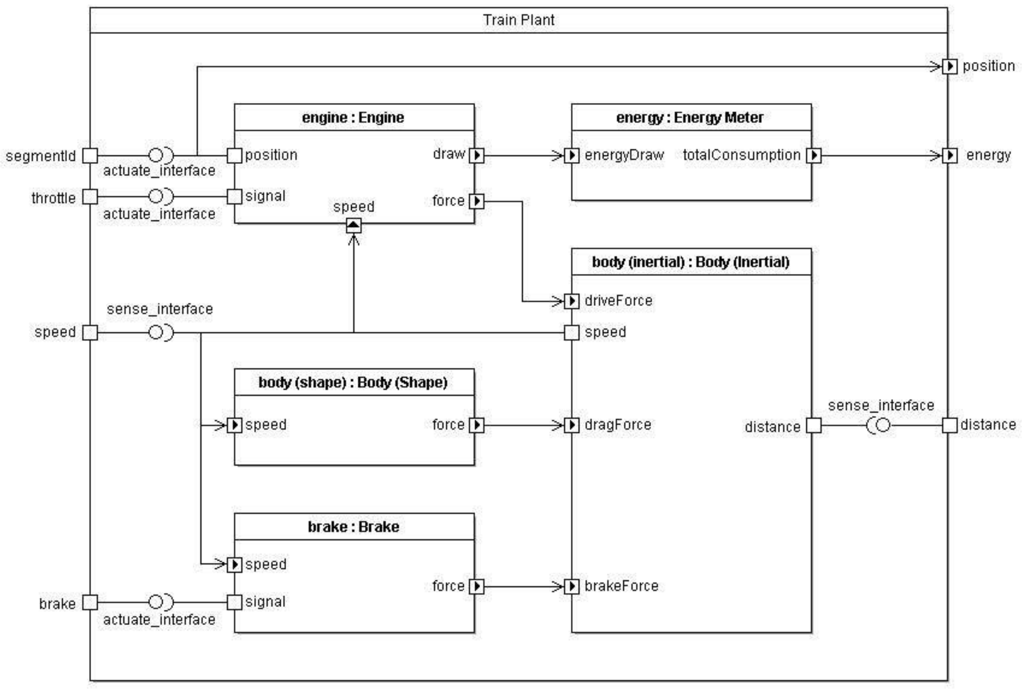


Figure 6.11: Composite System Connections View for Train Plant constituent

Each elementary (non-composite) system defined in this collection of structural diagrams is identified as either a *Physical System* or a *Cyber System*. Further specification of the properties and characteristics of these constituents is aided by the realisation of additional viewpoints.

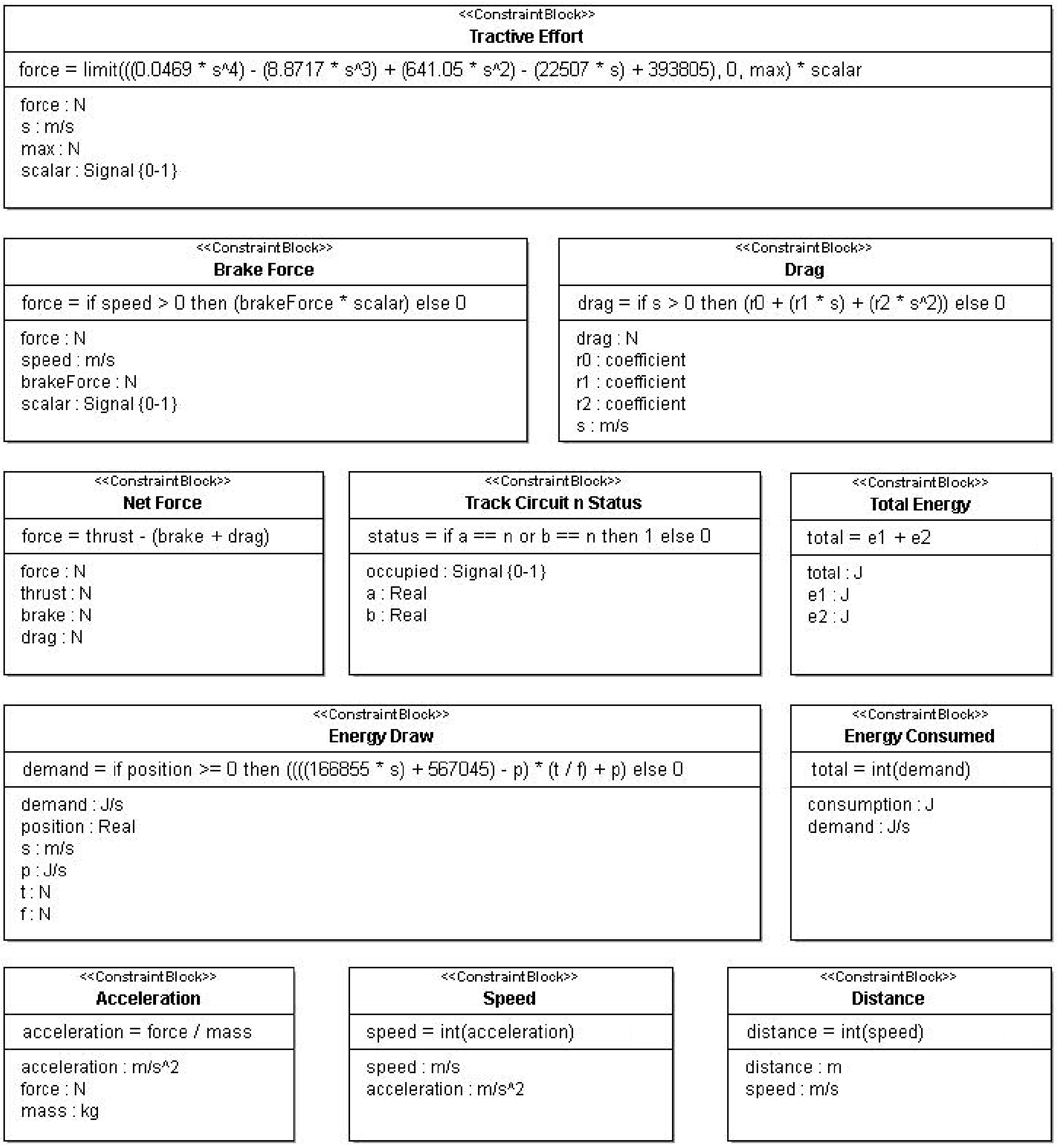
Physical Systems

Each elementary constituent indicated as a *Physical System* should be modelled using a continuous time formalism. A specification for the modelling of continuous time phenomena is encapsulated in an Equation Utilisation View (EUV), which must be constructed for each *Physical System* to satisfy the requirements of the architectural framework. Each EUV includes one or more continuous time expressions, and the architectural framework requires that these expressions are defined in an Equation Definition View.

SysML is used for the construction of an EDV including all expressions to be used in subsequent EUVs in Figure 6.12. For clarity, a single equation is specified for characterising the behaviour of a track circuit for train detection (*Track Circuit n Status*). In the model, this is replaced by a series of similar blocks, where in each case n is replaced by the identifier of the track segment in which the equation is used.

An EUV for the *Track Plant* constituent is described in Figure 6.13. For clarity, only a subset of train detection equations are shown. The model includes an explicit Track Circuit Status equation for each modelled block section. This constituent is responsible for the calculation of total energy used by all trains running on the infrastructure, and the indication of the presence of a train for each modelled block section.

Additional EUVs are specified for each of the subsystems of the *Train Plant* constituent. The *Energy Meter* constituent is responsible for calculating the total energy consumption of a train, and an EUV describing its dynamics is given in Figure 6.14. The *Body (Inertial)* constituent is responsible for calculating the movement of the train based on contributing forces, and an EUV describing its dynamics is given in Figure 6.15. The *Brake* constituent is responsible for calculating the force exerted by the application of breaks given a particular control signal, and an EUV describing its dynamics is given in Figure 6.16. The *Engine* constituent is responsible for calculating the force exerted by the action of the driving engine given a particular control signal, and an EUV describing its dynamics is given in Figure 6.17. The *Body (Shape)* constituent is responsible for calculating forces resisting the movement of the train resulting from its overall shape, and an EUV describing its dynamics is given in Figure 6.18.



```
<<ConstraintBlock>>
Tractive Effort
force = limit(((0.0469 * s^4) - (8.8717 * s^3) + (641.05 * s^2) - (22507 * s) + 393805), 0, max) * scalar
force : N
s : m/s
max : N
scalar : Signal {0-1}

<<ConstraintBlock>>
Brake Force
force = if speed > 0 then (brakeForce * scalar) else 0
force : N
speed : m/s
brakeForce : N
scalar : Signal {0-1}

<<ConstraintBlock>>
Drag
drag = if s > 0 then (r0 + (r1 * s) + (r2 * s^2)) else 0
drag : N
r0 : coefficient
r1 : coefficient
r2 : coefficient
s : m/s

<<ConstraintBlock>>
Net Force
force = thrust - (brake + drag)
force : N
thrust : N
brake : N
drag : N

<<ConstraintBlock>>
Track Circuit n Status
status = if a == n or b == n then 1 else 0
occupied : Signal {0-1}
a : Real
b : Real

<<ConstraintBlock>>
Total Energy
total = e1 + e2
total : J
e1 : J
e2 : J

<<ConstraintBlock>>
Energy Draw
demand = if position >= 0 then ((((166855 * s) + 567045) - p) * (t / f) + p) else 0
demand : J/s
position : Real
s : m/s
p : J/s
t : N
f : N

<<ConstraintBlock>>
Energy Consumed
total = int(demand)
consumption : J
demand : J/s

<<ConstraintBlock>>
Acceleration
acceleration = force / mass
acceleration : m/s^2
force : N
mass : kg

<<ConstraintBlock>>
Speed
speed = int(acceleration)
speed : m/s
acceleration : m/s^2

<<ConstraintBlock>>
Distance
distance = int(speed)
distance : m
speed : m/s
```

Figure 6.12: Equation Definition View for Rail Interlocking CPSoS

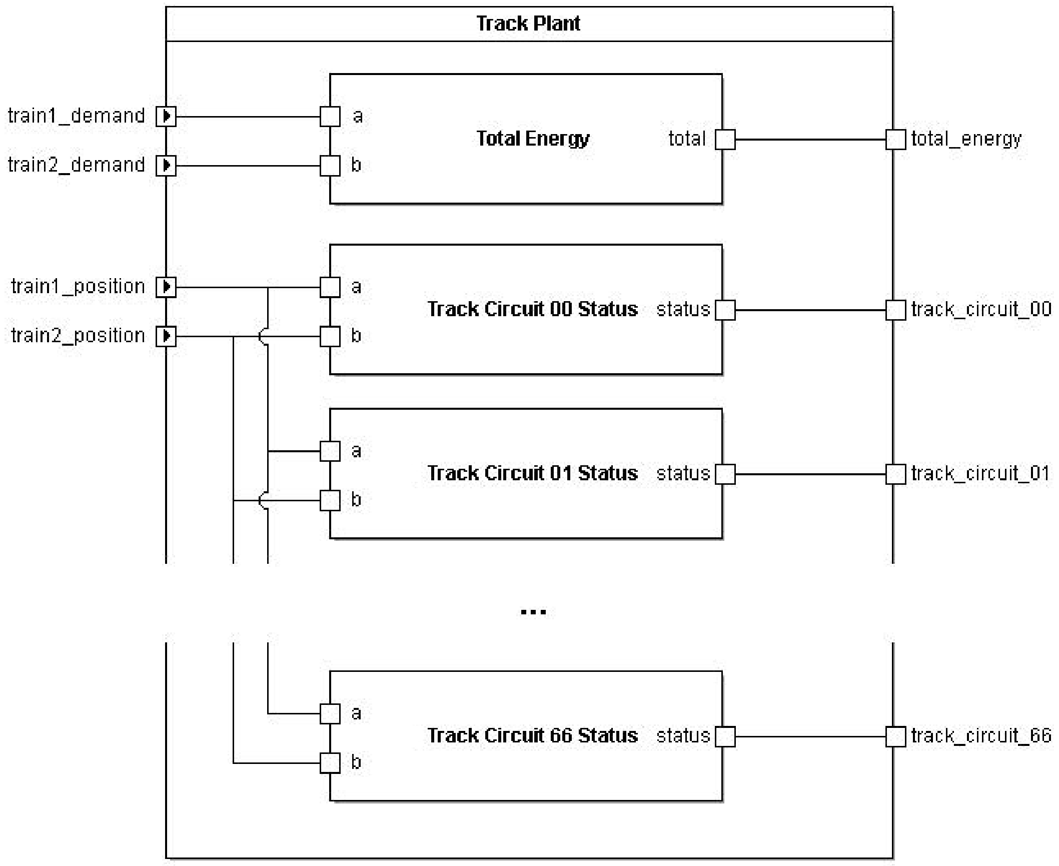


Figure 6.13: Equation Utilisation View for Track Plant Constituent

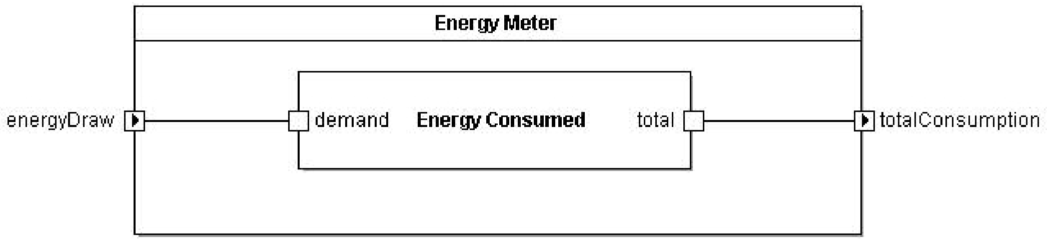


Figure 6.14: Equation Utilisation View for Energy Meter Constituent

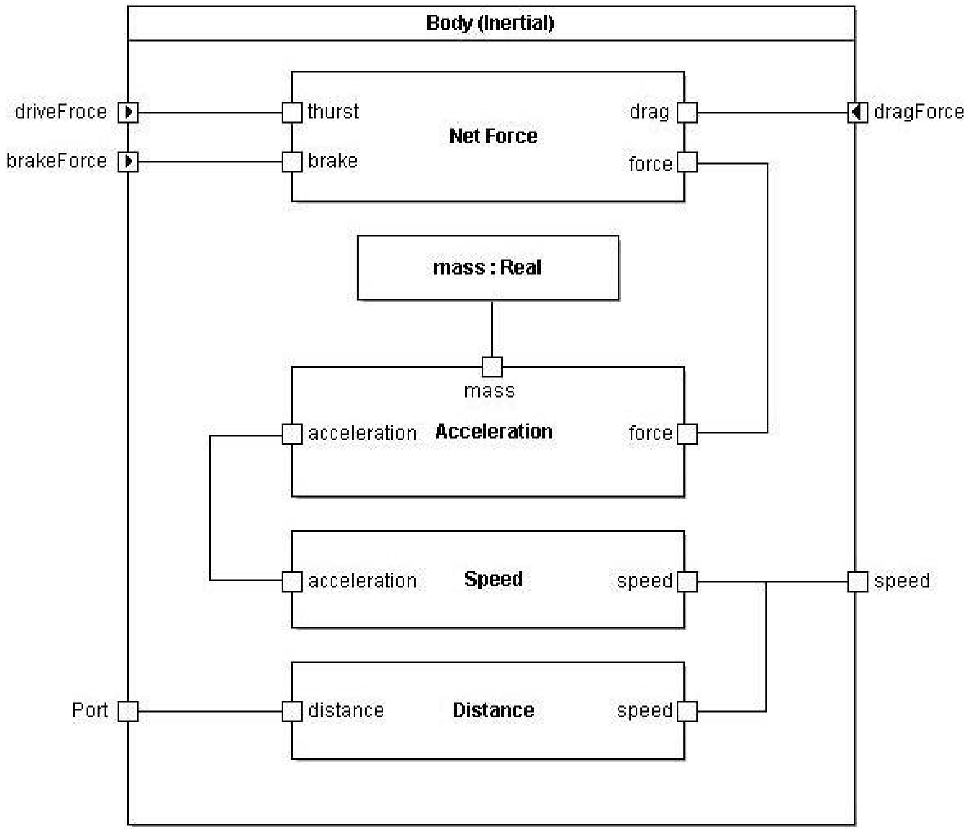


Figure 6.15: Equation Utilisation View for Body (Inertial) Constituent

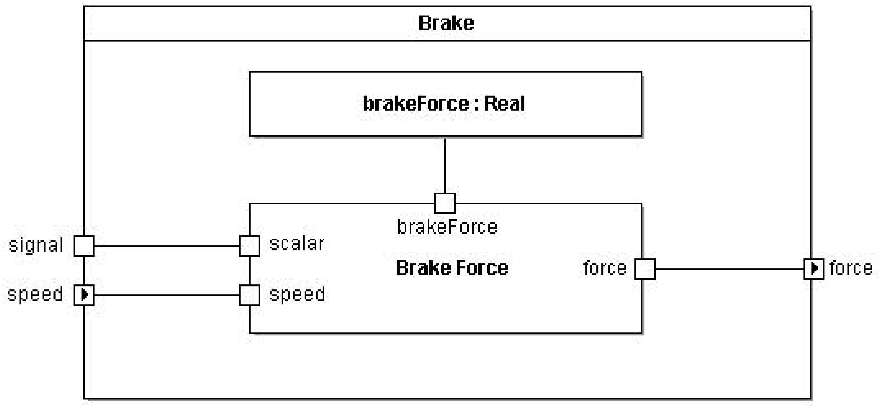


Figure 6.16: Equation Utilisation View for Brake Constituent

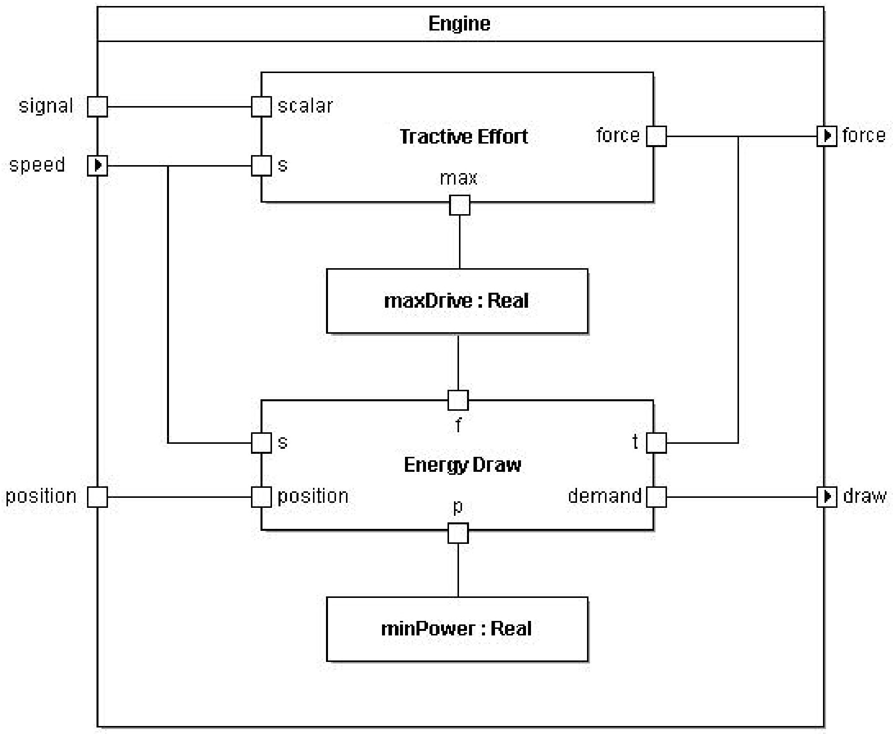


Figure 6.17: Equation Utilisation View for Engine Constituent

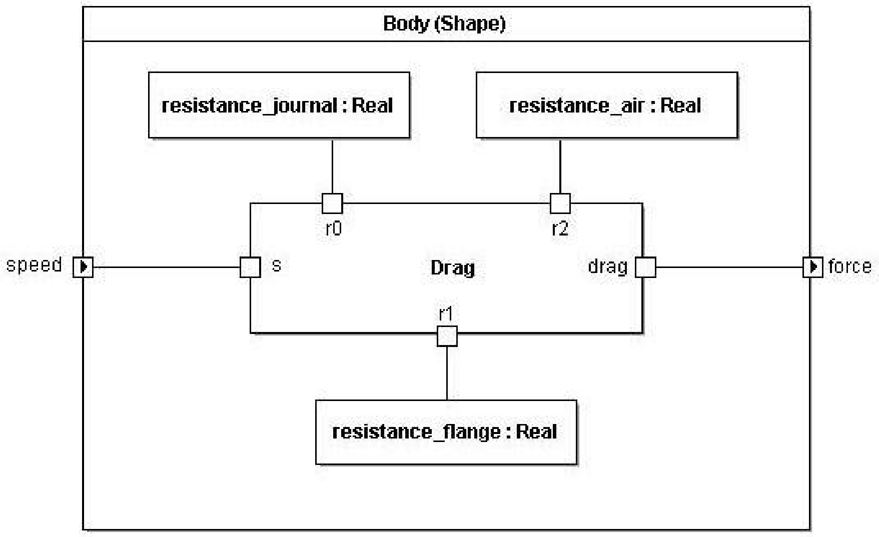


Figure 6.18: Equation Utilisation View for Body (Shape) Constituent

6.3.2 Co-Model Description

A co-model of the system is constructed based on the architectural description outlined in Section 6.3.1. This manually constructed co-model provides a specification for an automatically generated equivalent, and supports verification of the proposed approach to CPSoS engineering by enabling evaluation of the model dynamics through co-simulation.

Continuous Time Model

For each composite constituent system, a 20-sim *composite submodel* is created. Within each composite submodel, an *equation submodel* is created for each *Physical System*, and populated with equations describing its continuous time dynamics. For each *Cyber System*, an additional equation submodel is created and populated with mechanisms for linking the CT model to a DE model. An interface is defined for each submodel, and submodels are connected as specified in the architectural description.

The rail interlocking system is composed of three constituent composite systems. A composite submodel is created for each, and the models are connected, as illustrated in Figure 6.19.

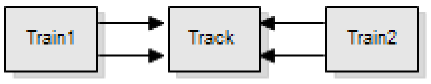


Figure 6.19: 20-sim Composite Submodels for Rail Interlocking CPSoS

The *Track* constituent is composed of two elementary subsystems, and an equation submodel is created for each, as illustrated in Figure 6.20. The *Track Plant* submodel describes physical dynamics, outlined in Listing 6.1, whilst the *Track Controller* submodel describes interaction with a DE model, outlined in Listing 6.2. In each case, only a subset of equations for train detection are included to aid clarity.

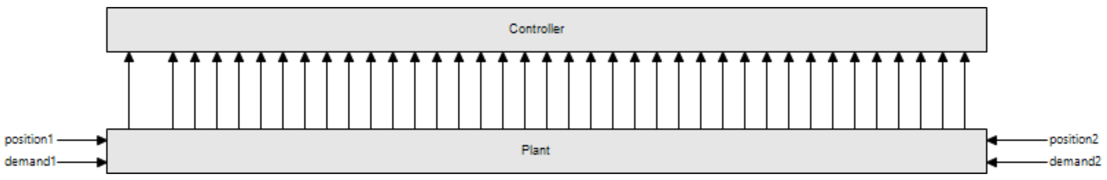


Figure 6.20: 20-sim Composite Submodels for Track Constituent

```
equations
  demand = demand1 + demand2;

  tc10 = if position1 == 0 or position2 == 0 then 1 else 0 end;
  tc11 = if position1 == 1 or position2 == 1 then 1 else 0 end;
  ...
  tc65 = if position1 == 65 or position2 == 65 then 1 else 0 end;
  tc66 = if position1 == 66 or position2 == 66 then 1 else 0 end;
```

Listing 6.1: 20-sim Equation Submodel for Track Plant

```
externals
  real global export track_demand;
  real global export tc1_status;
  real global export tc2_status;
  ...
  real global export tc65_status;
  real global export tc66_status;


initialequations
  track_demand = 0.0;
  tc0_status = 0.0;
  tc1_status = 0.0;
  ...
  tc65_status = 0.0;
  tc66_status = 0.0;

equations
  track_demand = demand;
  tc0_status = tc0;
  tc1_status = tc1;
  ...
  tc65_status = tc65;
  tc66_status = tc66;
```

Listing 6.2: 20-sim Equation Submodel for Track Controller

Each *Train* constituent is composed of two subsystems, and a submodel is created for each, as illustrated in Figure 6.21. An elementary submodel is created for the *Train Controller* constituent, describing interaction with a DE model, outlined in Listing 6.3. A composite submodel is created for the *Train Plant*, illustrated in Figure 6.22. Since the *Train1* and *Train2* constituents are identical, only one description is given, but the model includes two identical instances of this.

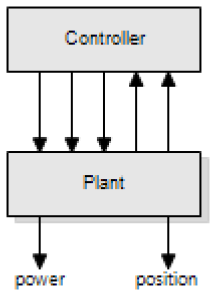


Figure 6.21: 20-sim Composite Submodels for Train Constituent

```
externals
  real global import train1_throttle;
  real global import train1_brake;
  real global import train1_position;
  real global export train1_speed;
  real global export train1_distance;

initialequations
  train1_speed = 0.0;
  train1_distance = 0.0;

equations
  throttle = train1_throttle;
  brake = train1_brake;
  position = train1_position;
  train1_speed = speed;
  train1_distance = distance;
```

Listing 6.3: 20-sim Equation Submodel for Train Controller

The *Train Plant* submodel includes five elementary constituents, each describing some part of the train dynamics. Each of these is described in an equation submodel, outlined in Listings 6.4–6.8.

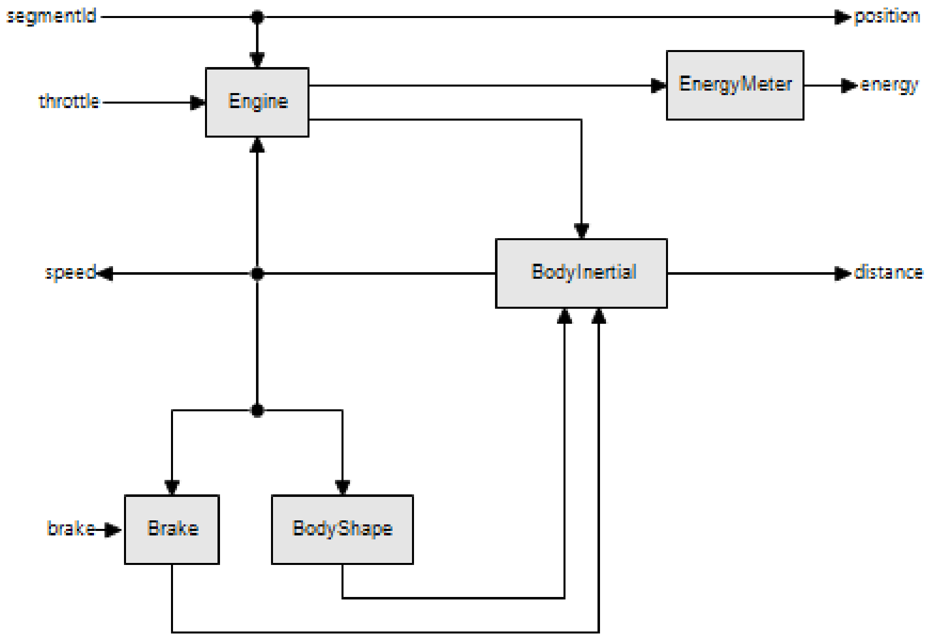


Figure 6.22: 20-sim Composite Submodels for Train Plant Constituent

```
parameters
real maxForce = 160000;
real minPower = 174000;

equations
thrust = limit(((0.0469*speed^4) - (8.8717*speed^3) + (641.05*speed^2) -
(22507*speed) + 393805),0,maxForce)*throttle;

demand = if position < 0
then 0
else ((((166855*speed)+567045)-minPower)*(thrust/maxForce))+ minPower
end;
```

Listing 6.4: 20sim Equation Submodel for Engine

```
equations
  totalConsumption = int(energyDraw);
```

Listing 6.5: 20sim Equation Submodel for Energy Meter

```
parameters
  real MaxForce = 147247;

equations


  force = if speed > 0
          then MaxForce*brake_value
          else 0
          end;
```

Listing 6.6: 20-sim Equation Submodel for Brake

```
parameters
  real journal_resistance = 3222;
  real flange_resistance = 112.8;
  real air_resistance = 7.802;

equations
  drag = if speed > 0
         then (journal_resistance + (flange_resistance * speed)
          + (air_resistance * speed^2))
         else 0
         end;
```

Listing 6.7: 20-sim Equation Submodel for Body (Shape)

```
variables
real force;
real acceleration;

equations
force = (drive - braking) - drag;
acceleration = force / mass;
speed = int(acceleration);
distance = int(speed);
```

Listing 6.8: 20-sim Equation Submodel for Body (Inertial)

Discrete Event Model

The DE model comprises a $System$ class containing instances of all controllers, sensors and actuators. The definition of controllers, sensors, and actuators for the rail interlocking CPSoS is given in Listing 6.9. Whilst the model includes a train detection sensor object for each modelled block section, only a subset of these is included in the listing for clarity.

```
system System
  instance variables
    public static train1_controller : [TrainController] := nil;
    train1_throttle : Throttle;
    train1_brake : Brake;
    train1_speed : Speedometer;
    train1_odometer : Odometer;
    train1_position: Position;
    train1_route : Route;

    public static train2_controller : [TrainController] := nil;
    train2_throttle : Throttle;
    train2_brake : Brake;
    train2_speed : Speedometer;
    train2_odometer : Odometer;
    train2_position: Position;
    train2_route : Route;

    public static track_control : [TrackController] := nil;
    track_demand : Voltmeter;
    tc0 : TrackCircuit;
    tc1 : TrackCircuit;
    ...
    tc65 : TrackCircuit;
    tc66 : TrackCircuit;
...
```

Listing 6.9: Rail Interlocking Cyber Constituents Definition

Virtual CPUs are created for each DE constituent. Where interfaces are specified for the interaction between distinct DE constituent, virtual busses are created and used to link virtual CPUs. The definition of virtual CPUs and busses is described in Listing 6.10.

```
...
    train1_CPU : CPU := new CPU(<FP>, 1E5);
    train2_CPU : CPU := new CPU(<FP>, 1E5);
    track_CPU : CPU := new CPU(<FP>, 1E5);

    bus1 : BUS := new BUS(<CSMACD>, 72E3,{train1_CPU, track_CPU});
    bus2 : BUS := new BUS(<CSMACD>, 72E3,{train2_CPU, track_CPU});
...
```

Listing 6.10: Case Study System Virtual Networking

Controllers are initialised by the provision of references to any owned sensor and actuator instances, and deployed on a virtual CPU. Instantiation of controllers is described in Listing 6.11

```
...
  operations
    public System : () ==> System
    System () == (
      train1_speed := new Speedometer();
      train1_odometer := new Odometer();
      train1_throttle := new Throttle();
      train1_brake := new Brake();
      train1_position := new Position();
      train1_route :=
        [10,11,12,13,14,15,16,30,31,32,33,34,35,36,40,41,42,43,44,45,46];

      train1_controller := new TrainController (
        train1_speed,
        train1_odometer,
        train1_throttle,
        train1_brake,
        train1_position,
        train1_route
      );

      train1_CPU.deploy(train1_controller);
      train2_speed := new Speedometer();
      train2_odometer := new Odometer();
      train2_throttle := new Throttle();
      train2_brake := new Brake();
      train2_position := new Position();
      train2_route :=
        [57,56,55,54,53,52,51,50,35,34,33,32,31,27,26,25,24,23,22,21,20];

      train2_controller := new TrainController (
        train2_speed,
        train2_odometer,
        train2_throttle,
        train2_brake,
        train2_position,
        train2_route
      );

      train2_CPU.deploy(train2_controller);

      track_demand := new Voltmeter();
      tc0 := new TrackCircuit();
      tc1 := new TrackCircuit();
      tc65 := new TrackCircuit();
      tc66 := new TrackCircuit();

      track_control := new TrackController(
        track_demand,
        tc0,
        tc1,
        ...
        tc65,
        tc66
      );

      track_CPU.deploy(track_control);
    );
end System
```

Listing 6.11: Case Study System Initialisation

Co-Simulation Contract

VDM and 20-sim models are linked by a co-simulation contract. Illustrated in Listing 6.12, the co-simulation contract maps shared variables to local instances.

```
input train1_speed = System.train1_speed.value;
input train1_distance = System.train1_odometer.value;
output train1_throttle = System.train1_throttle.value;
output train1_brake = System.train1_brake.value;
output train1_position = System.train1_position.value;

input train2_speed = System.train2_speed.value;
input train2_distance = System.train2_odometer.value;
output train2_throttle = System.train2_throttle.value;
output train2_brake = System.train2_brake.value;
output train2_position = System.train2_position.value;

input track_demand = System.track_demand.value;
input tc0_status = System.tc0.value;
input tc1_status = System.tc1.value;
...
input tc65_status = System.tc65.value;
input tc66_status = System.tc66.value;
```

Listing 6.12: Case Study Co-model contract

6.3.3 Co-Simulation Results

Co-simulation of the co-model can demonstrate the fidelity of the models of train dynamics, and the effectiveness of the implemented train control procedures. Basic control logic is added to the DE model to regulate the speed of each train, and to enable the authorisation of movement authorities. To demonstrate a critical requirement for interlocking for the given scenario, this initial model does not include route locking mechanisms. Co-simulation of the model provides a rich set of results, including the block occupied by each train throughout the simulation, illustrated in Figure 6.23a. The graph highlights that whilst the model is able to successfully simulate both modelled routes, the two movements are conflicting. In the absence of collision detection, the conflict is demonstrated by the shared occupation of *block 33* between $t \simeq 440$ and $t \simeq 500$.

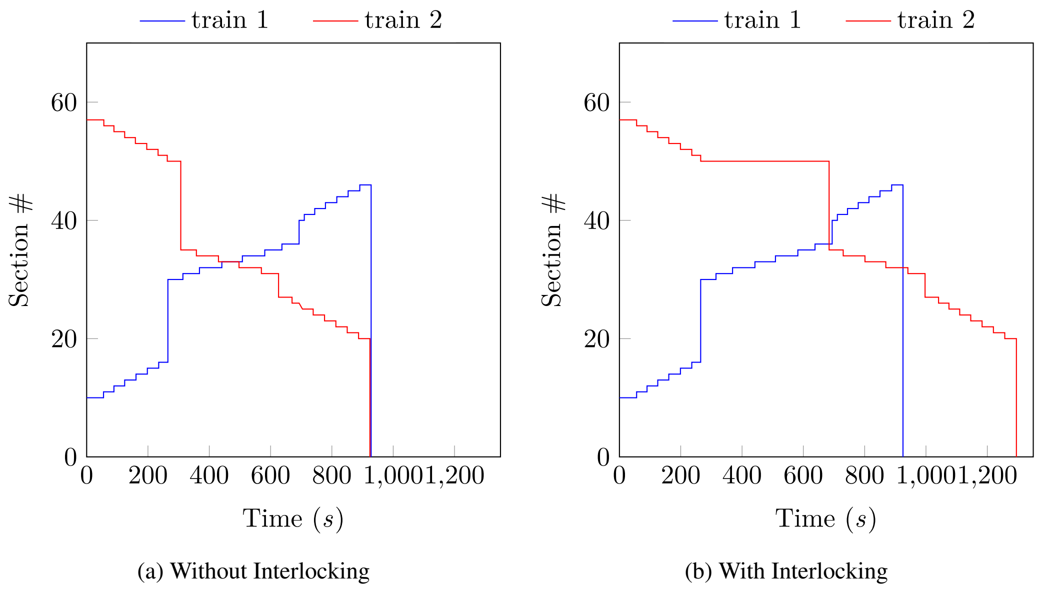


Figure 6.23: Co-Simulation Results: Route Progression

To ensure safe use of the modelled bridge, track sections must be locked for only one route, with the other route held in a safe block section until the shared resource again becomes available. After augmenting the DE model by the addition of control logic for the interlocking of resources, co-simulation results confirm the effectiveness of the implemented interlocking policy. Results from the simulation of the safe management of the two modelled routes are illustrated in Figure 6.23b. In this simulation, *train 1* is granted authorisation to cross the bridge, and *train 2* is held two is held until *train 1* has cleared any interlocked sections.

This demonstrates the effectiveness of the model for checking safety cases. Results from the co-simulation can clearly identify any unsafe movements by calculating the relative position of trains. The position of trains is calculated by high fidelity continuous time models of their dynamics. The dynamic models include consideration of both the speed of each train and the energy used to actuate the speed. An example of this is illustrated in Figure 6.24, which shows the calculated speed and energy draw of *train 2* throughout the simulation.

Four significant milestones in the simulated progression of the train are illustrated:

$t \simeq 0$: The train accelerates towards the bridge. Once at the maximum permitted speed, the train stops accelerating, and cruises at the permitted speed. By provision of a detailed representation of the acceleration and braking capabilities of the train, the results demonstrate how the model can be used to investigate the impact of driving strategies on energy use.

$t \simeq 300$**:** The train comes to a complete stop. On approaching the bridge, the track is locked for the progression of *train 1*. Since interlocking the bridge requires action for the prevention of conflicting movements, movement authority to progress beyond *block 50* is denied until the interlocked sections are released.

$t \simeq 650$**:** The train enters the interlocked section. After *train 1* has cleared the bridge, the interlocked resources are released, before being locked for the waiting train. The train accelerates on to the bridge, before briefly braking to accommodate a reduced permitted speed restriction over the interlocked section.

$t \simeq 1000$**:** The train exits the interlocked section. After clearing the speed restrictions of the bridge sections, the train accelerates to a greater permitted speed.

Further to the detailed representation of train movements, co-simulation results can also be used to assess the impact of strategies for the control of trains and the interlocking of resources on both the throughput of traffic and the total energy consumption of trains across the network. Figure 6.25 illustrates both the movement of rolling stock under the coordination of the implemented control strategies, and the resulting impact of those movement on the total consumption of energy.

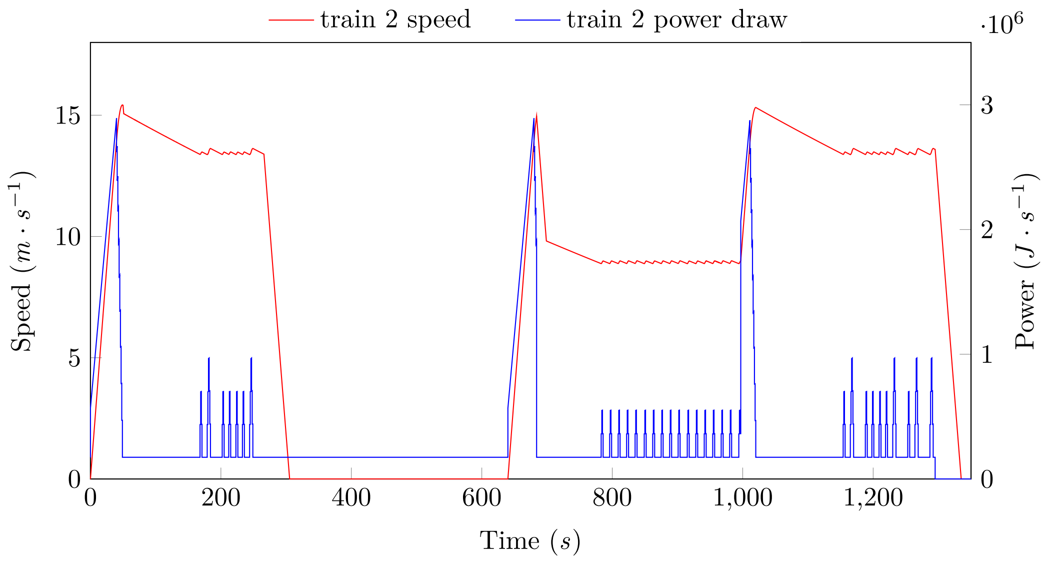


Figure 6.24: Co-Simulation Results: Train Dynamics

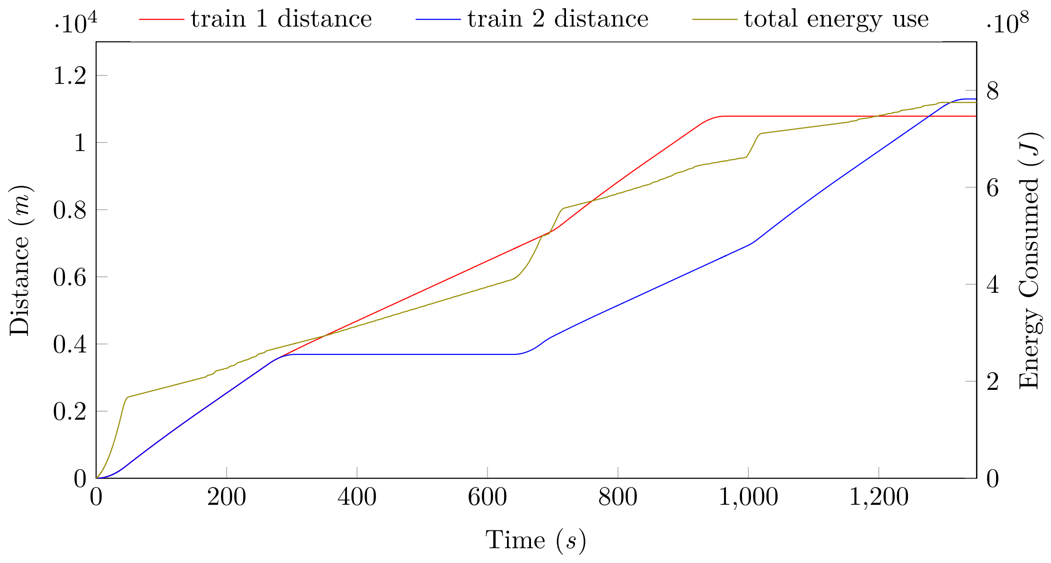


Figure 6.25: Co-Simulation Results: Traffic Throughput and Total Energy Demand

6.4 Co-model Generation

VDM-SL can be used to build an abstract representation of both the SysML and co-model representations of the rail interlocking case study. The construction of abstract representation of the underlying meta-models can be used to verify the co-model translation process defined in Chapter 5.

Section 6.4.1 describes the construction of abstract meta-models, by the instantiation of meta-model constructs in VDM-SL. In Section 6.4.2, the co-model meta-model is used as a specification for the expected output of the translation process, given the SysML meta-model as an input. Comparison of the actual and expected results of the translation process is used to verify the translation.

6.4.1 Meta-model Specification

Verification of the co-model generation process requires the VDM-SL representation of the meta-models underpinning both the SysML model of the rail interlocking scenario and the corresponding co-model.

The construction of VDM-SL representations of each meta-model is undertaken by the mapping of concrete models to the VDM-SL constructs defined in Chapter 5.

SysML Model

The SysML meta-model is constructed by the instantiation of a $CompositeContent$ type, as described in Listing 6.13.

```
sysml_model : SysMLModel = mk_CompositeContent(
  interlockingChildren,
  interlockingConnections
)


interlockingChildren : map Id to Block = {
  'Train1' |-> train1Block,
  'Train2' |-> train2Block,
  'Track' |-> trackBlock
}


interlockingConnections :  set of Flow = {
  mk_ItemFlow(
    't1position',
    mk_PortReference('Train1','position'),
    mk_PortReference('Track','train1_position')
  ),
  mk_ItemFlow(
    't1power',
    mk_PortReference('Train1','power'),
    mk_PortReference('Track','train1_power')
  ),
  mk_ItemFlow(
    't2position',
    mk_PortReference('Train2','position'),
    mk_PortReference('Track','train2_position')
  ),
  mk_ItemFlow(
    't2power',
    mk_PortReference('Train2','power'),
    mk_PortReference('Track','train2_power')
  )
}
```

```
train1Block : Block = mk_Block(
  {
    'position' |-> mk_FlowPort(<_out>),
    'power' |-> mk_FlowPort(<_out>)
  },
  mk_CompositeContent(train1Children,train1Connections)
)


train2Block : Block = mk_Block(
  {
    'position' |-> mk_FlowPort(<_out>),
    'power' |-> mk_FlowPort(<_out>)
  },
  mk_CompositeContent(train2Children,train2Connections)
)


trackBlock : Block = mk_Block(
  {
    'train1_position' |-> mk_FlowPort(<_in>),
    'train1_power' |-> mk_FlowPort(<_in>),
    'train2_position' |-> mk_FlowPort(<_in>),
    'train2_power' |-> mk_FlowPort(<_in>)
  },
  mk_CompositeContent(trackChildren,trackConnections)
)
```

Listing 6.13: Abstract descritpion of Rail Interlocking Model: Composite Structures

Each of the *Blocks* defined in Listing 6.13 is also composed of type *CompositeContent*, so this definition process is applied recursively for all composite child blocks.

For each *Block* of type *NodeDescription*, a *CyberNode* or *PhysicalNode* type is instantiated, as appropriate. The instantiation of a *CyberNode* is demonstrated in Listing 6.14, by the description of the *TrackController* constituent, and the instantiation of a *PhyiscalNode* type is demonstrated in Listing 6.15, by the description of the *TrackPlant* constituent.

```
trackPlant : PhysicalNode = mk_PhysicalNode(
  trackPlantConstraints
  {|->},
  trackPlantBindings
)

trackPlantConstraints : map Id to Constraint = {
  'Total Energy' |-> mk Constraint(-, 'a','b','total'),
  'Track Circuit 00 Status' |-> mk Constraint(-, 'a','b','status'),
  'Track Circuit 01 Status' |-> mk Constraint(-, 'a','b','status'),
  ...
  'Track Circuit 65 Status' |-> mk Constraint(-, 'a','b','status'),
  'Track Circuit 66 Status' |-> mk Constraint(-, 'a','b','status')
}

trackPlantBindings : set of Binding = {
  mk_Binding('total_energy',mk_paramterRef('Total Energy','total')),
  mk_Binding('train1_demand',mk_paramterRef('Total Energy','a')),
  mk_Binding('train2_demand',mk_paramterRef('Total Energy','b')),
  mk_Binding('train1_position',mk_paramterRef('Track Circuit 00 Status','a')),
  mk_Binding('train2_position',mk_paramterRef('Track Circuit 00 Status','b')),
  mk_Binding('track_circuit_00',mk_paramterRef('Track Circuit 00 Status','status')),
  ...
  mk_Binding('train1_position',mk_paramterRef('Track Circuit 66 Status','a')),
  mk_Binding('train2_position',mk_paramterRef('Track Circuit 66 Status','b')),
  mk_Binding('track_circuit_66',mk_paramterRef('Track Circuit 66 Status','status'))
}
```

Listing 6.14: Abstract descritpion of Rail Interlocking Model: Physical Systems

```
trackController : CyberNode = {
  mk_ControllerClass('TrackController',-)
}
```

Listing 6.15: Abstract descritpion of Rail Interlocking Model: Digital Systems

Co-model

Described in Listing 6.16, the co-model meta-model is constructed by the instantiation of a *CoModel* type, a composition of an instantiated *CTModel* and an instantiated *DEModel* type.

```
comodel : CoModel = mk_CoModel(
  ctModel,
  deModel
)
```

Listing 6.16: Abstract descritpion of Rail Interlocking Model: Co-Models

A *CTModel* is constructed by the instantiation of a *CompositeModel* type, as described in Listing 6.17.

```
ctModel : CompositeModel = mk_CompositeModel(
  interlockingSubmodels,
  comodelConnections
)

interlockingSubmodels : map Id to submodel(
  'Train1' |-> train1Submodel,
  'Train2' |-> train2Submodel,
  'Track' |-> trackSubmodel
)

comodelConnections :  set of Connection = {
  mk_Connection(
    't1position',
    mk_PortReference('Train1','position'),
    mk_PortReference('Track','train1_position')
  ),
  ...
}
```

```
train1Submodel : Submodel = mk_Submodel(
  mk_CompositeModel(train1Submodels,interlockingCoModelConnections)
  {
    'position' |-> mk_SubmodelPort(<_output>,<_signal>),
    'power' |-> mk_SubmodelPort(<_output>,<_signal>)
  }
)
```

Listing 6.17: Abstract descritpion of Rail Interlocking Model: CT Model Structure

Similarly to the definition of the SysML meta-model, the definition of composite constructs is applied recursively for all submodels.

For each $Submodel$ of type $ElementaryModel$, a $PhysicalModel$ or $ControlLinkModel$ type is instantiated, as appropriate. The instantiation of a $PhysicalModel$ is demonstrated in Listing 6.14, by the description of the $TrackPlant$ constituent, and the instantiation of a $ControlLinkModel$ type is demonstrated in Listing 6.15, by the description of the $TrackController$ constituent.

```
trackPlantSubmodel : PhysicalModel = mk_PhysicalModel(
{|->},
{
  mk_Equation([Total Energy]),
  mk_Equation([Track Circuit 00 Status]),
  ...
  mk_Equation([Track Circuit 66 Status])
}
)
```

Listing 6.18: Abstract descritpion of Rail Interlocking Model: Physical Model

```
trackControlLink : PhysicalModel = mk_ControlLinkModel(
{
  'totalDemand' |-> <_export>,
  'tc0' |-> <_export>,
  ...
  'tc66' |-> <_export>
}
)
```

Listing 6.19: Abstract descritpion of Rail Interlocking Model: Control Link Model

A *DEModel* is constructed by the instantiation of a *DEModel* type, as described in Listing 6.20. A DE model is a composition of a *VDMSystem* and a *Contract*, described in listings 6.21 and 6.22, respectively.

```
deModel : DEModel = mk_DEModel(
  vdmSystem,
  contract
)
```

Listing 6.20: Abstract descritpion of Rail Interlocking Model: DE Model

```
vdmSystem : VDMSystem = mk_VDMSystem(
  {
    'Train1_controller' |-> train1Controller,
    'Train2_controller' |-> train2Controller,
    'Track_controller' |-> trackController,
  },
  {
    mk_Bus('Train1_controller','Track_controller'),
    mk_Bus('Train2_controller','Track_controller'),
  }
)
```

Listing 6.21: Abstract descritpion of Rail Interlocking Model: DE Model

```
contract : Contract = {
  'train1_throttle' |-> <_controlled>,
  'train1_brake' |-> <_controlled>,
  ...
  'trackEnergyDemand' |-> <_monitored>
}
```

Listing 6.22: Abstract descritpion of Rail Interlocking Model: DE Model

6.4.2 Meta-model Comparison

The Overture tool for VDM modelling enables the automated evaluation of VDM expressions. This can be used to test for equality between the co-model meta-model defintion representing the manually contrstructed co-mode, and the meta-model produced by the application of the model transformation process to the SysML meta-model definition.

Confirmation of equality of the two meta-models provides confidence that the defined model mapping is valid.

6.5 Discussion and Evaluation

A rail interlocking system is developed to support the verification of the proposed architectural framework and supporting SysML profile, and the proposed co-model generation process.

The merit of the approach is evaluated by consideration of the extent to which the proposed approach satisfies the verification criteria outlines in Section 6.1.

6.5.1 Architectural Modelling

Verification criteria for the evaluation of the effectiveness of the approach include considerations for a sufficiently detailed architecture, and requirements for the realisation of architectural descriptions. The extent to which the proposed approach satisfies criteria for the specification of architectures is summarised in Table 6.5, and the extent to which the proposed approach satisfies criteria for the realisation of architectures is summarised in Table 6.6.

Table 6.5: CPSoS Architectural Modelling Criteria

Criteria	**Proposed Solution**
Terms and concepts important to the specification of a CPSoS must be defined, and the relationships between concepts identified.	By the inclusion of a CPSoS ontology, the proposed architectural framework explicitly facilitates the identification of CPSoS concepts.
A collection of important considerations of a system of interest must be specified	By the specification of a series of CPSoS viewpoints, the proposed architectural framework isolates the necessary considerations of a CPSoS architecture.

By demonstrating the application of the proposed techniques for the development of architectural descriptions of CPSoS, the rail interlocking case study provides confidence in the suitability of the approach.

The architectural framework enables the detailed description of the system, including consideration of system structures, constituent interfaces, and cyber and physical behaviours. The architectural framework is uniquely advantageous over alternative frameworks which might be employed for the description of the rail interlocking system by its provision of viewpoints which support the realisation of constituents in complementary formalisms.

Table 6.6: CPSoS Architecture Realisation Criteria

Criteria	**Proposed Solution**
For each element specified in an architectural description, expression of the element must be possible using an architectural definition language.	The proposed SysML profile augments SysML by the inclusion of concepts important to the realisation of CPSoSs, where concepts are identified by the CPSoS ontology included in the proposed architectural framework.
Guidelines for the use of an architectural definition language must explicitly relate architectural elements to language constructs	The definition of the proposed SysML profile includes a series of profile diagrams, which specify how profile constructs should be employed in the realisation of each viewpoint of the proposed architectural framework.

For the realisation of the system architecture according to the viewpoints of the proposed framework, the SysML profile provides effective mechanisms for the description of all necessary considerations of the system.

6.5.2 Co-Model and Co-Simulation

Verification criteria for the evaluation of the effectiveness of the approach includes requirements for the support of the production of executable models. The extent to which the proposed approach satisfies criteria for the production of executable models is summarised in Table 6.7.

By demonstrating the use of co-models for the simulation of CPSoSs, the rail interlocking case study provides confidence in the suitability of the approach.

Co-simulation demonstrates the capacity of the approach for the evaluation of factors which emerge from the interaction of a collection of independent constituents composed of both digital and dynamic components. Detailed simulation results demonstrate the advantage of supporting high fidelity models in complementary formalisms.

Since co-simulation is particularly costly, and less well suited to exhaustive analysis than alternative

Table 6.7: CPSoS Executable Model Criteria

Criteria	Proposed Solution
Executable models must utilise complementary formalisms to ensure high fidelity of modelled phenomena of both a continuous and discrete nature.	The nomination of heterogeneous co-models for the enabling co-simulation of CPSoSs ensures that descriptions of cyber and physical constituents can be separately modelled in complementary environments. By facilitating the description of constituents in complementary environements, the proposed approach supports high fidelity representations of both discrete and continuous phenomena.
Complementary abstractions of a system must be consistent.	The proposed approach advocates the description of CPSoSs using complementary abstractions. To ensure consistency between models, an automated translation between formalisms is specified. Whilst this does not facilitate the translation of concrete models, it does demonstrate a viable translation to motivate the implementation of the approach in a suitable tool.

static techniques. However, for the exploration of the dynamics of a given scenario (e.g. for optimisation), co-simulation facilitates uniquely detailed descriptions of CPSoS bahaviours.

The manual construction of abstract representations of the meta-models underpinning the complementary abstractions of the rail study enables verification of the transformation specification. Confirmation that the transformation produces suitable results for a given input by comparison with an expected output demonstrates that should an implementation respect the specified process, it will facilitate accurate translation of models.

6.6 Summary

Verification of the proposed approach to the engineering of CPSoSs is essential in the provision of confidence that it satisfies its requirements. Verification of the proposed techniques is undertaken by the application of the approach to an appropriate case study. To facilitate verification, criteria for the success of the proposed approach are established in Section 6.1, alongside criteria for an appropriate system to which the approach should be applied.

The chosen case study describes the interlocking of resources in a rail network. The rail system is a large and critical CPSoS with significant management and engineering challenges. An SoS approach to rail management is essential, but the optimisation of emergent properties such as throughput capacity and energy use demands detailed representation of the interaction of both digital and physical systems. A detailed description of the constituent systems of a rail interlocking system is given in Section 6.2, where data is taken from relevant literature and domain experts.

Models describing the chosen system are described in Section 6.3. Models include an architectural description, and a manually constructed co-model. The manually constructed co-model provides a specification to support the verification of a co-model description generated from the given architecture. Co-simulation of the co-model is used to verify that the model is capable of execution of the heterogeneous models derived from the architectural description. Verification of the translation process is outlined in Section 6.4, where comparison of the translated output and the manually constructed model provides confidence in the effectiveness of the co-model generation process.

Demonstration of the approach enables the identification of the strengths and weaknesses of the proposed techniques. These are evaluated by comparison to alternative model-based engineering approaches in Section 6.5. Verification of the proposed architectural framework with supporting profile, as well as the co-model generation process, demonstrates that the approach is appropriate to support the model-based engineering of CPSoSs, and is uniquely advantageous over viable alternatives.

7 Approach Validation

The utility of the proposed engineering approach can be assessed by *validation*. Validation describes the confirmation that a system or process meets the expectations of its intended use by the provision of objective evidence. Validation comprises a set of activities to provide confidence in the ability of a system or process is able to achieve its intended use in its intended environment (ISO/IEC/IEEE 15288 2015, p. 7). Whilst verification is aimed at ensuring a solution behaves as intended, validation is aimed at ensuring that a solution overcomes the intended challenges (Martin 1996).

The overall purpose of the proposed engineering approach is to support activities concerning the engineering of CPSoSs by employing existing modelling technologies for both CPS and SoS engineering. To satisfy this goal, the proposed techniques must utilise each of the selected CPS and SoS modelling technologies in such a way that they better facilitate the engineering of CPSoSs.

Existing model-based tools utilised by the approach include SysML for architectural modelling and Crescendo co-models for facilitating co-simulation. SysML is tailored for use in CPSoS engineering by the provision of an architectural framework and supporting language profile. For the approach to be effective, SysML must better support the engineering of CPSoSs when used in conjunction with the proposed framework and profile. Co-models are manipulated for use in CPSoS engineering by techniques for the automation of their creation, based on architectural models. An effective approach must also support the creation of co-models guided by an architectural specification.

Measuring the effectiveness of each technology for supporting the engineering of CPSoSs requires a suitable objective metric. The maturity of a particular technology for a specific application is described as *technology readiness*, and the assessment of technology readiness can be supported by examination of concepts, requirements, and demonstrated capabilities using an appropriate framework.

This chapter describes the development of an appropriate technology readiness framework, and its use in assessing the utility of the proposed engineering techniques. Section 7.1 describes the development of a suitable technology readiness framework for the assessment of modelling technologies for CPSoS engineering. In Section 7.2, the framework is used in the assessment of the suitability of SysML and co-modelling technologies for CPSoS engineering, both with and without the support of the proposed techniques. Comparison of the technology readiness of each modelling technology used with and without the proposed support is evaluated in Section 7.3.

7.1 Validation Framework

Technology Readiness Levels (TRLs) describe a metric to support the assessment of the maturity of a particular technology and the constituent comparison between different types of technology. The TRL approach was introduced by NASA in 1974 as a discipline-independent, programmatic figure of merit to facilitate more effective assessment of, and communication regarding the maturity of new technologies. (Mankins 1995).

The assignment of a TRL to a particular technology is facilitated by a Technology Readiness Assesment (TRA). TRAs are attempts to determine the maturity of a new technology, and might be undertaken throughout the development of a new technology, including following the completion of conceptual design, to aid selection from competing designs, and prior to committing to final development. TRAs might be small scale and informal, or a large, highly formal process involving a diverse range of stakeholders (Mankins 2009).

Whilst the original TRL metric comprised a seven-level scale for the categorisation of technology maturity, a revised nine-level scale developed in the 1990s gained widespread acceptance (Banke 2010). The NASA TRL scale ranges from TRL 1 ("basic principles observed and reported") to TRL 9 (actual system proven through successful operation"), and is used extensively for the categorisation of technologies throughout industry and academia. The scale is included in systems engineering standards since its publication in *ISO 16290 Definition of the Technology Readiness Levels (TRLs) and their criteria of assessment* (ISO 16290 2013), and is widely considered an effective mechanism for establishing a common understanding of technology status, for identifying risk associated with component immaturity, and optimising resource expenditure.

Despite considerable evidence to support the utility of TRLs, its suitability is limited for the assessment of nonphysical system components such as software. Originally designed for assessing the maturity of technologies for the development of space systems, the application of TRLs to software-based technologies is often challenging (Smith 2005, Graettinger et al. 2002), with Héder (2017) claiming that "the concreteness and sophistication of the TRL scale gradually diminished as its usage spread outside of its original context (space programs)".

The United States Department of Defence (DoD) was among the first to attempt to extend the TRL assessment approach to include definitions of software maturity, and the DoD Technology Readiness Assessment (TRA) Deskbook provides guidance for system and software definitions of technology maturity (US Department of Defence 2003). Based on the DoD guidelines for TRL assessment, the US Army Aviation and Missile Research, Development and Engineering Center (AMRDEC) provide an approach to the measuring of TRLs for software, described as the Software Readiness Level (SRL) assessment process. Similarly to TRLs, SRLs comprise nine levels of technology maturity but are tailored to describing the maturity of software (Graettinger et al. 2002).

During the COMPASS project[1], both TRLs and SRLs were employed in the evaluation of tools and methods for SoS engineering developed as part of the project. Application of each metric was considered an inadequate mechanism for effective evaluation of both tools and methods. In response, COMPASS developed two further mechanisms: a technology benchmarking process (Ingram et al. 2013), and a Method Readiness Levels (MRL) metric (Holt et al. 2014).

The MRL approach is an adaptation of the SRL approach, where level assessment criteria were remodelled to suit the evaluation of methods developed in the COMPASS project. The revised model was used extensively for the evaluation of COMPASS engineering approaches, where assessment is performed as a step-wise progression between levels. The levels are outlined in Table 7.1 (adapted from Holt et al. (2014)).

The COMPASS technology benchmarking approach includes a well-defined set of processes for benchmarking COMPASS tools against baseline technologies. For each of a series of engineering processes, the performance of a technology in supporting the process is assessed to establish a baseline capability. Evaluation of baseline technologies includes methods, implementation languages, and tools evaluation. The evaluation process is repeated for a new technology, and the same assessment is used to compare the effectiveness of the technology with the baseline.

COMPASS technology benchmarking is used to evaluate the effectiveness of COMPASS tools when applied to the engineering of two SoS case studies. For each case study, utilisation of baseline technologies for overcoming engineering challenges is evaluated (Forcolin et al. 2013, Kristensen et al. 2013). The same engineering challenges are approached using COMPASS technologies, and the process is evaluated (Casoto et al. 2014, Kristensen et al. 2014), and comparison of the two benchmarking evaluations

[1]See `http://www.compass-reasearch.eu/`

Table 7.1: COMPASS Method Readiness Levels

Level	Description	Supporting evidence
1: Basic principles observed	Basic source information is gathered and collated as an input to demonstrating need	Academic/industrial papers, surveys
2: Technology concept or application formulated	Concept is defined through definition of needs Model or Method	Requirement Description Views, Context Definition Views, Requirement Context Views
3: Characteristic proof of concept	Validation criteria for concept is defined, analysed and reviewed	Example Views
4: Model or Method defined based on concepts and proof	The Model or Method of the approach, based on the concepts and proof is defined and verified	Approach (ontology, framework, etc) defined using established framework (eg seven views, CAFF or other framework)
5: Model or Method validated on relevant test applications	Partial Model or Method of approach is applied to one or more test applications, such as established or pre-defined test application Model or Methods	Incomplete set of artefacts (views) produced based on the defined approach using one or more test applications, generated by academic or industry partners
6: Model or Method demonstration in relevant environment	Model or Method of approach is completed and process is defined. Approach Model or Method is applied to one or more test applications, such as established or pre-defined test application Model or Methods	Complete set of artefacts (views) produced based on the defined approach using one or more test applications. Processes for Framework defined generated by academic or industry partners
7: Model or Method demonstration in operational environment	Complete Model or Method is applied to one or more industrial case studies	Complete set of artefacts (views) produced based on industry case study, generated by industry partners
8: Model or Method completed and qualified	Complete Model or Method tailored for specific industry and is applied on real industry projects	Tailored Model or Method developed and applied on real industry projects
9: Model or Method proven through successful mission operations	Model or Method becomes part of industry approach, is measured and controlled by industry quality system	Tailored Model or Method becomes part of industry quality management system

is used to demonstrate the utility of the features technologies.

Examples of COMPASS technology benchmarks include:

SoS Requirement Stakeholder Identification. The model-based method must support identification and definition of SoS requirement stakeholders. Stakeholders determine the requirements. Stakeholders must be mapped to one or more requirements.

SoS Emergent Behaviours Requirement. The model-based method must support development of emergent behaviours requirement that conform to standard requirements quality attributes. An EBRA must be consistent with other requirements, testable and traceable to contributing constituent systems.

SoS Model Verification. The model-based method and modelling language must provide guidelines and modelling capabilities for simulation of use-cases, and sound SoS reasoning. Simulation results must be decidable.

SoS Architectural Modelling. The modelling language must provide SoS modelling capabilities for developing optimal architecture.

SoS Test Model Development. The model-based test method must provide guidelines, architectural patterns and modelling capabilities for development of SoS test models.

SoS Scalability Testing. The model-based method must support SoS scalability modelling. The scalability of test models must be testable.

SoS Model Identification. The model-based method and SysML must provide aggregation and modelling capabilities allowing identification and incorporation of SoS elements and architectural patterns.

SoS Fault Tolerance Analysis. The model-based method should provide excellent support and guidance of the analysis of fault tolerance, by helping to identify possible faults, events that trigger them and supporting the development of recovery strategies.

In evaluating both baseline and new technologies, technologies are assigned a suitability score for each benchmark. A suitability score describes the suitability of the technology for fulfilling the requirements of the particular process being examined. The range of suitability scores are in summarised in Table 7.2.

Technology benchmarking can be used to gain confidence that the utilisation of proposed methods and tools can positively contribute to overcoming particular engineering challenges. Whilst the approach is intended to provide objective evidence to support claims of merit, the validity of benchmark scoring is limited by the inherent subjectivity of any capability assessment.

Table 7.2: COMPASS Benchmarking Suitability Scored

Score	Suitability
0	Insufficient
1	Poorly corresponds to expectations
2	Able to perform the required task
3	Performs the required task well
4	Significantly more effective than alternative methods

7.1.1 Technology Benchmarking

Both the COMPASS technology benchmarking approach and MRLs can be used in the evaluation of the utility of the proposed techniques for supporting the engineering of CPSoSs.

The benchmarks introduced technology evaluation are developed exclusively for the assessment of technologies for engineering SoSs, but can be modified for the assessment of technologies for engineering CPSoSs.

Whilst only a subset of the COMPASS technology benchmarks are applicable for the assessment of CPSoS engineering techniques, additional benchmarks are required for the evaluation of the proposed approach. A benchmarking framework for the evaluation of technologies to aid the engineering of CPSoSs can be derived from the approach objectives outlined in Chapter 1. Benchmarks to be used in the assessment of the suitability of technologies for supporting the engineering of CPSoSs include:

Architectural Modelling. CPSoS development is dependent on the organisation of many distinct elements, where systems are composed of independent constituents. A suitable architectural language must facilitate the description of hierarchically organised systems, and must be independent of any particular engineering domain.

Model Identification. Model-based techniques must provide aggregation capabilities allowing identification of constituents, and the designation of an appropriate formalism for the representation of constituent dynamics.

Model Development. An engineering process must include guidelines and patterns for the development of CPSoS models.

Interface Analysis. A model-based approach to CPSoS engineering should provide support for the identification and description of interfaces between system components.

Discrete Event Phenomena Description. CPSoS models must include suitable mechanisms for the expression of behaviours of a discrete event nature.

Continuous Time Phenomena Description. CPSoS models must include suitable mechanisms for the expression of behaviours of a continuous time nature.

Evaluation of Emergent Behaviours. The overall behaviour of a CPSoS is the result of interactions between autonomous constituents, where individual behaviours are the result of interactions between cyber and physical components. Analysis of these emergent behaviours demands dynamic models with the capacity to simulate constituents and their interactions.

Model Consistency. The model-based development of CPSoSs demands the integration of diverse models, and the representation of system components at different levels of abstraction. Complementary descriptions of system components must be consistent in their representation of the component.

These benchmarks will be used in Section 7.2 for the assessment of the suitability of technologies for aiding the engineering of CPSoSs.

7.2 Benchmark Assessment

Technology benchmarks can be used to assess the suitability of a particular technology for a particular function. The proposed techniques for model-based engineering of CPSoSs utilise existing technologies commonly used for both SoS engineering and CPS engineering, specifically, SysML and Crescendo co-models. Assessment of the suitability of each of these existing technologies for the engineering of CPSoSs can provide a baseline against which the utility of the proposed approach can be measured.

By the evaluation of both existing and proposed technologies using the same benchmarking framework, the impact of the proposed techniques can be assessed. Further evaluation of the utility of the proposed approach can be facilitated by consideration of the method readiness of both baseline technologies and the proposed techniques.

Technology benchmark scores and readiness estimations are assigned for the baseline technologies in Section 7.2.1, and scores for technologies used in conjunction with the proposed techniques are assigned in Section 7.2.2.

7.2.1 Baseline Technologies

SysML

SysML is a general-purpose modelling language for systems engineering. It supports high-level specification of hierarchical system architectures, and includes mechanisms for the expression of discrete systems inherited from UML, and additional mechanisms for the expression of continuous time phenomena by the introduction of mechanisms such as parametric diagrams. Evaluation of SysML for CPSoS modelling by consideration of the proposed technology benchmarks is summarised in Table 7.3.

There is extensive support for the use SysML in the engineering of SoSs, including patterns and frameworks. The applicability of these guidelines for the engineering of CPSoSs is limited, with consideration of the characteristics of constituent systems typically limited to digital systems. There is a well-established need to extend the use of SysML for the engineering of CPSoSs but there is little support for doing so. Given this, the use of SysML in the engineering of CPSoSs is assigned an MRL of 1.

Cresendo Co-models

Crescendo co-models combine the VDM modelling language for the abstraction of software systems, and the 20-sim modelling environment for the abstraction of physical dynamics. By supporting co-modelling of discrete event and continuous time phenomena in distinct environments, crescendo models enable high fidelity abstractions of both cyber and physical system elements. The Crescendo co-simulation engine facilitates execution of co-models, where separate discrete event and continuous time simulators are progressed in parallel, and data can be shared between simulators to facilitate model interaction. Evaluation of co-models for CPSoS modelling by consideration of the proposed technology benchmarks is summarised in Table 7.4.

There is extensive support for the use of co-models in the engineering of CPSs, including patterns and frameworks. The applicability of these guidelines for the engineering of CPSoSs is limited, with little or no consideration of interaction outside of the system boundary, or autonomy of system components. There is a well-established need for the use of heterogenous simulation is the engineering of CPSoSs but there is little support for the use of co-models in this way. Given this, the use of co-models for the engineering of CPSoSs is assigned an MRL of 1.

7.2.2 Proposed Technologies

Architectural Framework

The proposed architectural framework provides guidance for the construction of architectural models of CPSoSs by the provision of a set of viewpoints. A supporting SysML profile restricts and extends SysML to tailor its use for realisation of the architectural framework viewpoints. Utilisation of the profile to realise the framework provides a systematic approach for the description of CPSoSs using SysML. Evaluation of the proposed architectural framework and supporting language profile for CPSoS modelling by consideration of the proposed technology benchmarks is summarised in Table 7.5.

The proposed architectural framework and supporting SysML profile guides the use of SysML for the production of architectural descriptions of CPSoSs. The approach is defined using an established framework, and demonstrated using a case study. Given this, the proposed architectural framework for the engineering of CPSoS is assigned an MRL of 5.

Co-model Generation

The proposed process for co-model generation provides a specification for the automated generation of co-models based on an architectural description of a CPSoS. The process demonstrates that the automated generation of co-models is plausible given a sufficiently detailed architectural description, where sufficient detail is ensured by the use of the proposed architectural framework. The purpose of the specification is to test the feasibility of such a translation, though it is not implemented in a supporting tool. Verification of the model translation specification in Chapter 6 provides confidence that the process is effective, and by benchmarking the translation as if it were implemented in an appropriate tool, the impact of the translation can be demonstrated to motivate such an implementation. Given this, evaluation of co-model generation for CPSoS modelling by consideration of the proposed technology benchmarks is summarised in Table 7.6.

The proposed co-model generation process demonstrates the viability of an automated translation between complementary abstractions of a CPSoS. By the automation of co-model construction, the use of co-models for the engineering of CPSoSs is assinged an MRL of 3.

Table 7.3: SysML Benchmark Scores

Architectural Modelling	**4**
SysML enables the description of system elements using generic *blocks*. Blocks can be specialised by the use of stereotypes, making it extensible for the description of elements from any engineering domain.	
Model Identification	**2**
SysML includes mechanisms for the hierarchical organisation systems including mechanisms for describing relationships such as composition and aggregation. SysML does not include specific syntax for the expression of properties such as implementation formalism, but stereotypes can be used to tailor generic blocks.	
Model Development	**0**
SysML is independent of any application, and does not include guidelines or patterns.	
Interface Analysis	**3**
SysML supports the definition of interfaces by the use of ports and connectors. Inherited from UML, ports can expose a particular digital interface, and specialised flow ports represent the transfer of physical quantities between components. Internal Block Diagrams facilitate can be used to model interfaces and connections between modelled components.	
Discrete Event Phenomena Description	**4**
SysML inherits mechanisms for the expression of digital elements from UML. Class diagrams can be used to construct detailed expressions of digital systems.	
Continuous Time Phenomena Description	**3**
By the addition of parametric diagrams, SysML supports the representation of the dynamics of physical systems.	
Evaluation of Emergent Behaviours	**0**
SysML models are static and do not include semantics of execution.	
Model Consistency	**0**
SysML does not support evaluation of models in complementary formalisms.	

Table 7.4: Co-modelling Benchmark Scores

Architectural Modelling	**1**
VDM supports the organisation distinct digital elements by the provision of virtual execution environments. VDM is an abstract notation and facilitates the representation of any software system. 20-sim supports the organisation of models by the provision of *submodels*. 20-sim includes mechanisms for representing elements from a range of domains, as well as abstract mechanisms for domain-independent modelling. Architectural modelling of the entire system is fragmented between constituent models, and is less abstract than architectural modelling languages.	
Model Identification	**1**
VDM is supports object-orientation for modelling composite digital elements, and 20-sim supports hierarchical model structuring by the provision of *composite submodels*. By facilitating modelling in both VDM and 20-sim, elements can be modelled using an appropriate formalism, though the designation of an appropriate formalism must be decided outside of the modelling environment.	
Model Development	**1**
Crescendo includes a selection of patterns for common CPS engineering activities, but does not include patterns or guidelines with consideration of SoS challenges.	
Interface Analysis	**3**
VDM supports the representation of digital interfaces by the provision of networking mechanisms for virtual execution environments and remote method calls for communication between objects. 20-sim supports interface modelling by the provision of ports and connectors. The definition of interfaces between DE and CT elements is supported by the creation of a co-simulation contract.	
Discrete Event Phenomena Description	**4**
VDM has a rich and abstract syntax for the representation of software-based systems, with support for object-orientation and real-time systems.	
Continuous Time Phenomena Description	**4**
20-sim has a rich and abstract syntax for the representation of physical systems of a wide range of engineering domains, and further supports domain-independent modelling.	
Evaluation of Emergent Behaviours	**4**
Co-simulation supports the execution of heterogeneous models, coordinating the progression of separate DE and CT simulators and facilitating the transfer of data between environments.	
Model Consistency	**0**
Co-models do not support evaluation of models in complementary formalisms.	

Table 7.5: Architectural Framework Benchmark Scores

Criterion	Score
Architectural Modelling	**4**
The architectural framework ensures that a complementary set of descriptions includes the description of any system components important to the description of a CPSoS. The SysML profile enables the specialisation of *Blocks* to explicitly identify composite systems and ensure their decomposition.	
Model Identification	**4**
The architectural framework includes a Composite System Structure Viewpoint to ensures that components are explicitly assigned an appropriate implementation formalism. The SysML profile defines stereotypes to facilitate the assignment of a model type to any non-composite systems.	
Model Development	**4**
The architectural framework guides the production of models by the specification of CPSoS-specific viewpoints. The SysML profile augments SysML by the inclusion of terms and concepts related to CPSoS engineering to guide the realisation of architectural descriptions, and tailors generic SysML elements for the expression of CPSoS concepts.	
Interface Analysis	**4**
The architectural framework includes a Composite System Connections Viewpoint to ensure that interfaces between constituents are explicitly modelled. The SysML profile specialises the use of SysML ports and connectors to describe the nature of interactions between diverse components.	
Discrete Event Phenomena Description	**3**
The architectural framework ensures that digital elements are identified but does not provide guidance for their construction. The use of UML/SysML for the description of digital systems is well established, and the proposed techniques to not augment existing solutions.	
Continuous Time Phenomena Description	**4**
The architectural framework ensures that digital elements are decomposed and their behaviour modelled. Equation Definition Viewpoint and Equation Utilisation Viewpoint ensure that the behaviour of dynamic systems is well-defined in architectural systems. The SysML profile guides the realisation of these viewpoints using SysML Parametric Diagrams and *Constraint Blocks*.	
Evaluation of Emergent Behaviours	**4**
The use of the SysML profile to realise views to satisfy the architectural framework guides the production of architectural descriptions which can be translated to co-models. By automation of this refinement, the augmentation of architectural models with the addition of co-simulation semantics enables simulation of a model specified at an architectural level.	
Model Consistency	**4**
By the provision of a mapping between complementary abstractions of, CPSoS models refinement can be automated. Automation of model refinement ensures a systematic approach to the preservation of semantics across system views.	

Table 7.6: Co-Model Generation Benchmark Scores

Criterion	Score
Architectural Modelling	**4**
The organisation of model elements is abstractly defined outside of the co-modelling environment. This systematic approach to co-model construction affords the abstract expressiveness of SysML in the hierarchical organisation of co-model elements.	
Model Identification	**4**
The designation of an appropriate formalism for the modelling of each component is defined outside of the co-modelling environment. Composite DE and CT models are automatically populated with elements most appropriate to each environment.	
Model Development	**4**
The development of co-models is entirely automated based on an architectural description.	
Interface Analysis	**4**
Component interfaces and connections are defined outside of the co-modelling environment. This ensures that all required interfaces are created in the co-model.	
Discrete Event Phenomena Description	**3**
VDM has a rich and abstract syntax for the representation of software-based systems, with support for object-orientation and real-time systems.	
Continuous Time Phenomena Description	**4**
Continuous time dynamics are represented in 20-sim equation models which are automatically populated based on SysML descriptions.	
Evaluation of Emergent Behaviours	**4**
Co-simulation supports the execution of heterogeneous models, coordinating the progression of separate DE and CT simulators and facilitating the transfer of data between environments.	
Model Consistency	**4**
By the provision of a mapping between complementary abstractions of, CPSoS models refinement can be automated. Automation of model refinement ensures a systematic approach to the preservation of semantics across system views.	

7.3 Approach Impact

7.3.1 Architectural Modelling

A domain-independent and general-purpose modelling language, SysML is an ideal candidate for the construction of architectural descriptions of CPSoSs. Whilst it does not include syntax for the expression of CPSoS concepts, extensibility features such as inheritance and stereotypes enable the tailoring of SysML to support the model-based engineering of CPSoSs.

Summarised in Table 7.7, a baseline assessment of SysML indicates that it is well suited to CPSoS modelling but lacks domain-specific guidelines. Furthermore, SysML models are static and do not facilitate simulation of models, limiting the approach for the exploration of emergent behaviours. SysML models are abstract and often used in the development of more concrete representations, however as a standalone technology, SysML does not facilitate consistency checking with complementary abstractions.

Table 7.7: Technology Benchmark Comparison

Benchmark	**Baseline**	**Proposed**
Architectural Modelling	4	4
Model Identification	2	4
Model Development	0	4
Interface Analysis	3	4
Discrete Event Phenomena Description	3	3
Continuous Time Phenomena Description	3	4
Evaluation of Emergent Behaviours	0	4
Model Consistency	0	4

The proposed architectural framework and supporting SysML profile advance the suitability of SysML for the engineering of CPSoSs. The provision of viewpoints and guidelines for their realisation ensures systematic use of SysML, and the introduction of stereotypes tailors the language to include concepts important to CPSoS engineering. Whilst SysML models do not facilitate simulation, the co-model generation process further advances the suitability of SysML for CPSoS engineering, facilitating automated refinement of architectural descriptions to enable simulation.

Further to advancing the suitability of SysML, by employing the proposed architectural framework, the MRL of SysML modelling for CPSoS development is significantly increased. This demonstrates the utility of the framework and suggests that it is an effective mechanism for supporting the model-based engineering of CPSoSs.

7.3.2 Enabling Simulation

By enabling co-simulation of heterogenous systems, Crescendo co-models are an ideal candidate for the simulation of CPSoSs. Whilst co-models do not support the development of systems architectures, their hierarchical structure is well suited to the structuring of CPSoS elements.

Summarised in Table 7.8, a baseline assessment of co-modelling indicates that it is well suited to CPSoS simulation but lacks support for the organisation of constituent elements. Co-models are well suited to the implementation of architectural specifications, but do not facilitate consistency checking with complementary abstractions.

Table 7.8: Technology Benchmark Comparison

Benchmark	**Baseline**	**Proposed**
Architectural Modelling	1	4
Model Identification	1	4
Model Development	1	4
Interface Analysis	3	4
Discrete Event Phenomena Description	4	4
Continuous Time Phenomena Description	4	4
Evaluation of Emergent Behaviours	4	4
Model Consistency	0	4

The proposed co-model generation processing advances the suitability of co-models for the engineering of CPSoSs. By automating the construction of co-models based on rich architectural descriptions, the organisation of constituent components can be undertaken using SysML, and the abstract descriptions enhanced by the addition of co-simulation semantics. Co-simulation of CPSoSs enables exploration of properties that emerge from interaction between constituents and across the cyber-physical boundary.

Further to advancing the suitability of co-models, by their automated generation, the MRL of co-modelling for CPSoS development is increased. Since the co-model generation process is a specification for a model refinement, the enhancement of the readiness of co-models is limited. The specification demonstrates the utility of an implementation of the translation process, and motivates its implementation in a suitable tools framework.

7.4 Summary

Validation of the proposed approach to model-based development of CPSoSs provides objective evidence of the utility of the approach. Technology benchmarking and technology readiness assessments provide a structured approach to the evaluation of the proposed techniques, by consideration of the merit of the approach in comparison to baseline technologies.

In Section 7.1 a framework the evaluation of the proposed techniques is developed, based on existing mechanisms for the evaluation of technologies, tools, and methods. In Section 7.2 this framework is applied to establish the suitability of both existing technologies and the proposed approach for the engineering of CPSoSs.

In Section 7.3 the utility of the approach is assessed by comparison of the proposed techniques against baseline technologies.

By validation of the proposed approach, both the architectural framework and co-model generation process are demonstrated to improve the suitability and readiness of baseline technologies for the engineering of CPSoSs. The utility of the proposed techniques is limited until implemented in an appropriate tools framework, but demonstrated merit of the approach provides strong motivation for such an implementation.

8 Conclusions and Future Work

This thesis describes and evaluates a model-based approach to support the engineering of CPSoSs. CPSoSs are increasingly underpinning the next generation of critical infrastructures, with prominent examples in transport, manufacturing and buildings. The coordination of autonomous constituents comprising vast networks of both computational and physical devices is essential in responding to global challenges by improving energy and resource efficiency, reducing emissions and waste, and providing improved services at lower costs.

The engineering of CPSoSs is uniquely challenging, demanding mechanisms for the organisation of independently owned and managed components, and the close coordination of cyber and physical worlds. The systematic use of models in the design and development of CPSoSs is essential for overcoming these challenges, including the consideration of complementary views of the system at various levels of abstraction.

This Chapter summarises the approach by consideration of its strengths and weaknesses in Section8.1, and considers opportunities for its further development and exploitation in Section 8.2.

8.1 Conclusions

The proposed approach supports organisation of constituent systems using an architectural framework, supported by a SysML profile, to guide the realisation of architectural descriptions. The proposed framework ensures the decomposition of constituent systems to identify an appropriate domain for more concrete representations, and detailed specification of system dynamics. The architectural framework is unique in its support for the identification of target formalisms, and for the description of continuous time phenomena.

Additionally, the proposed approach demonstrates a mapping from architectural descriptions to more concrete, executable models. Discrete and continuous phenomena are expressed using complementary formalisms in a collaborative modelling environment, to ensure high fidelity representations of all constituents. Simulation of these models enables the analysis of complex properties and behaviours which emerge as a result of interaction between the cyber and physical worlds, and between autonomous constituents. The mapping demonstrates that the architectural framework ensures sufficiently detailed architectural descriptions to support the automated generation of executable models, providing a specification for model refinement by the addition of simulation semantics.

To ensure that the proposed techniques are effective in addressing weaknesses in the state of the art in model-based techniques for CPSoS engineering, verification of the approach is undertaken by the application of the techniques to a representative system of interest. Described in Chapter 6, verification confirms that the proposed architectural framework is adequate for the description of an appropriate system of interest in sufficient detail to support the automated generation of a corresponding co-model, by its application to a rail interlocking case study.

To demonstrate the utility of the approach, the proposed techniques are validated by consideration of their impact in advancing baseline technologies. Described in Chapter 7, technology benchmarks are proposed for the evaluation of CPSoS technologies, alongside the use of established mechanisms for the evaluation of technology readiness. Validation of the approach demonstrates the suitability of the proposed techniques for advancing candidate technologies in their capacity to support CPSoS engineering.

The research described in this thesis includes the design of an engineering approach, however the proposed techniques lack tool support. Without tool support, there is limited scope for assessing the utility of the proposed techniques. Tool support is essential in enabling additional verification and validation of the approach by facilitating further exercising of the techniques by their application to a variety of engineering challenges and their use by experienced practitioners. However, tool development is costly, and is limited in its contribution to the definition of the approach.

8.2 Future Work

Identified in Section 8.1, the embedding of the proposed engineering techniques in a suitable open tools framework is essential for its exploitation and further assessment of its suitability and merit. An appropriate tool chain for the implementation of the proposed techniques is provided by the INTO-CPS project[1]. The INTO-CPS tool chain integrates well established technologies to enable comprehensive model-based design of CPSs, supporting multidisciplinary, collaborative modelling from requirements through to realisation in hardware and software (Bagnato et al. 2015).

The INTO-CPS tool chain combines complementary modelling tools and formalisms to enable holistic modelling and simulation of CPSs by use of the domain-independent Functional Mockup Interface (FMI) (Blochwitz et al. 2011). Simulation of FMI-compliant models is facilitated by an extended co-simulation engine to enable interaction between a diverse set of models. Whilst the Crescendo co-simulation engine supports the co-simulation of a single 20-sim and VDM coupling, the INTO-CPS co-simulation engine enables the simulation of multiple instances of models developed by a variety of tools.

The INTO-CPS tool chain is supported by a SysML profile for the structuring of models and allocation of an appropriate implementation language (Bagnato et al. 2016). The profile does not satisfy an architectural framework, and does not support the decomposition of constituents or specification of behaviours. The use of the architectural framework and supporting SysML profile described in this thesis within the INTO-CPS tool chain would enhance its suitability for the engineering of CPSoSs.

INTO-CPS models can include constituent models developed in both 20-sim and VDM, as well as a variety of complementary formalisms, such as the open-source 20-sim alternative, OpenModelica[2]. Extension of the proposed translation process could evaluate the suitability of model generation for a given architectural description, or suggest further framework views of SysML profile mechanisms to enable the generation of models in a wider selection of formalisms.

The use of models in the engineering of complex systems is often employed to overcome challenges outside of the typical operation of a system, such as response to faults or malicious use. The proposed architectural framework does not provide viewpoints for explicit consideration of factors contributing to extenuating circumstances, but its augmentation to include such mechanisms would further advance its utility. Extension of the framework would include additional viewpoints such as those included in the Fault Modelling Architectural Framework (FMAF) (Andrews et al. 2013).

[1]See `http://into-cps.au.dk`
[2]See `https://openmodelica.org/`

8.3 Summary

By the provision of an architectural framework and supporting SysML profile, together with a specification for the automated generation of co-models based on resulting architectural descriptions, the proposed approach demonstrably advances the state of the art in model-based techniques to support the engineering of CPSoSs.

The utility of the approach can be significantly improved by its inclusion in an appropriate tool chain, and promising extensions to the approach include support for the automated generation of models in additional formalisms, and the inclusion of viewpoints for the modelling of issues such as fault tolerance and security.

The merit of the approach is demonstrated by consideration of specific improvements to the suitability and maturity of candidate technologies for supporting the model-based engineering of CPSoSs. This work provides a strong motivation for inclusion of the proposed techniques in an appropriate tool chain, and their use in the development of CPSoSs.

Bibliography

Abbott, R. (2006), Open at the top; open at the bottom; and continually (but slowly) evolving, *in* '2006 IEEE/SMC International Conference on System of Systems Engineering', IEEE, pp. 41–46.

Abrial, J.-R. (1996), *The B Book - Assigning Programs to Meanings*, Cambridge University Press, Cambridge, UK.

Acheson, P., Dagli, C. & Kilicay-Ergin, N. (2013), 'Model based systems engineering for system of systems using agent-based modeling', *Procedia Computer Science* **16**, 11–19.

Albrecht, C. & Reimann, M. (2013), 'Report on commonalities in the four domains and recommendations for strategic action', EC Road2SoS Project Deliverables D5.1 and D5.2; Online `http://road2sos-project.eu/cms/upload/documents/Road2SoS_D5.1_D5.2_CommonalitiesandRecommendations.pdf`. Accessed June 2016.

Alexander, I. F. & Beus-Dukic, L. (2009), *Discovering requirements: how to specify products and services*, John Wiley & Sons, Chichester, UK.

Allen, R. B. & Garlan, D. (1992), A Formal Approach to Software Architectures, *in* 'Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing 1992', North-Holland Publishing Co., Amsterdam, The Netherlands, pp. 134–141.

Allen, R. & Garlan, D. (1997), 'A Formal Basis for Architectural Connection', *ACM Transactions on Software Engineering and Methodology (TOSEM)* **6**(3), 213–249.

Allen, R. J. (1997), A Formal Approach to Software Architecture, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

Alur, R., Courcoubetis, C., Henzinger, T., Ho, P., Nicollin, X., Olivero, A., Sifakis, J. & Yovine, S.

(1994), The algorithmic analysis of hybrid systems, *in* '11th International Conference on Analysis and Optimization of Systems: Discrete-event Systems', Springer Verlag, pp. 331–351.

Amiry, A. (1965), The simulation of information flow in a steelmaking plant, *in* S. Hollingdale, ed., 'Digital Simulation in Operational Research', English University Press, London, UK, pp. 347–356.

Andrews, Z., Fitzgerald, J., Payne, R. & Romanovsky, A. (2013), Fault modelling for systems of systems, *in* 'Proceedings of the 11th International Symposium on Autonomous Decentralised Systems', IEEE Computer Society, pp. 59–66.

ANSI/EIA 632 (2003), 'Processes for Engineering a System', *American National Standards Institute / Electronic Industries Association* .

Ashenden, P. J. (2010), *The designer's guide to VHDL*, Vol. 3, Morgan Kaufmann, Burlington, MA, USA.

Augarten, S. (1983), The Most Widely Used Computer on a Chip: The TMS 1000, *in* 'State of the Art: A Photographic History of the Integrated Circuit', Ticknor & Fields, New Haven and New York, NY, USA.

Axelrod, R. (1997), *The complexity of cooperation: Agent-based models of competition and collaboration*, Vol. 3, Princeton University Press, Princeton, NJ, USA.

Bae, K., Ölveczky, P. C., Feng, T. H. & Tripakis, S. (2009), Verifying Ptolemy II Discrete-Event Models Using Real-Time Maude, *in* 'International Conference on Formal Engineering Methods', Springer, pp. 717–736.

Bagnato, A., Brosse, E., Quadri, I. R. & Sadovykh, A. (2015), 'INTO-CPS: An integrated "tool chain" for comprehensive model-based design of cyber-physical systems', *Revue Génie Logiciel* **113**, 31–35.

Bagnato, A., Brosse, E., Quadri, I. & Sadovykh, A. (2016), 'SysML for Modeling Co-simulation Orchestration over FMI: the INTO-CPS Approach', *Ada User Journal* **37**(4), 215–218.

Bahill, A. T. & Gissing, B. (1998), 'Re-evaluating systems engineering concepts using systems thinking', *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **28**(4), 516–527.

Baldwin, W. C. & Sauser, B. (2009), Modeling the characteristics of system of systems, *in* 'IEEE International Conference on System of Systems Engineering (SoSE) 2009', IEEE, pp. 1–6.

Banke, J. (2010), 'Technology readiness levels demystified', `https://www.nasa.gov/topics/aeronautics/features/trl_demystified.html`. Accessed June 2016.

Banks, J., Carson, J., Nelson, B. L. & Nicol, D. (2004), *Discrete-event System Simulation*, 4th edn, Prentice Hall, Upper Saddle River, NJ, USA.

Bekič, H. & Jones, C. B. (1984), Programming languages and their definition, *in* 'Lecture Notes in Computer Science (LNCS)', Vol. 177, Springer, Berlin, Germany.

Beynon-Davies, P. & Holmes, S. (2002), 'Design breakdowns, scenarios and rapid application development', *Information and software technology* **44**(10), 579–592.

Biggs, B. (2005), 'Ministry of defence architectural framework (MODAF)', *IEE Digest Seminar Series* **43**.

Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Elmqvist, H., Junghanns, A., Mauß, J., Monteiro, M., Neidhold, T., Neumerkel, D. et al. (2011), The functional mockup interface for tool independent exchange of simulation models, *in* 'Proceedings of the 8th International Modelica Conference', Linköping University Electronic Press, Linkping, Sweden, pp. 105–114.

Boardman, J., Pallas, S., Sauser, B. & Verma, D. (2006), Report on system of systems engineering, *in* 'Final Report for the Office of Secretary of Defense', Stevens Institute of Technology, Hoboken, NJ.

Boardman, J. & Sauser, B. (2006), System of Systems – the meaning of "of", *in* '2006 IEEE/SMC International Conference on System of Systems Engineering', IEEE, pp. 118–123.

Bonoma, T. V. (1985), 'Case research in marketing: Opportunities, problems, and a process', *Journal of Marketing Research* **22**(2), 199–208.

Börger, E. & Stärk, R. (2003), *Abstract state machines: a method for high-level system design and analysis*, Springer-Verlag, Berlin, Germany.

Breunese, A. P. & Broenink, J. F. (1997), 'Modeling Mechatronic Systems Using the SIDOPS+ Language', *Simulation Series* **29**, 301–306.

Broenink, J. F. (1999), '20-sim software for hierarchical bond-graph/block-diagram models', *Simulation Practice and Theory* **7**(5-6), 481–492.

Broenink, J. F., Ni, Y. & Groothuis, M. A. (2010), On model-driven design of robot software using co-simulation, *in* 'Proceedings of SIMPAR 2010 workshops international conference on simulation, modeling, and programming for autonomous robots', Technische Universität Darmstadt, Darmstadt, Germany, pp. 659–668.

Brooks, C., Cataldo, A., Lee, E. A., Liu, J., Liu, X., Neuendorffer, S. & Zheng, H. (2005), Hyvisual: A hybrid system visual modeler, Technical report, University of California, Berkeley, CA, USA.

Brooks, R. J. & Tobias, A. M. (1996), 'Choosing the best model: Level of detail, complexity, and model performance', *Mathematical and computer modelling* **24**(4), 1–14.

Brown, S. F. (2009), Naivety in systems engineering research: are we putting the methodological cart before the philosophical horse, *in* '7th Annual Conference on Systems Engineering Research (CSER 2009)', Loughborough University, Loughborough, UK.

Broy, M. (2013), Engineering cyber-physical systems: Challenges and foundations, *in* M. Aiguier, Y. Caseau, D. Krob & A. Rauzy, eds, 'Complex Systems Design & Management', Springer, Berlin, Heidelberg, pp. 1–13.

Bryans, J. W., Fitzgerald, J. S. & McCutcheon, T. (2011), Refinement-based techniques in the analysis of information flow policies for dynamic virtual organisations, *in* L. Camarinha-Matos, A. Pereira-Klen & H. Afsarmanesh, eds, 'Adaptation and Value Creating Collaborative Networks, IFIP Advances in Information and Communication Technology', Springer Boston, pp. 314–321.

Bryman, A. (1984), 'The debate about quantitative and qualitative research: a question of method or epistemology?', *British Journal of Sociology* **35**, 75–92.

Buede, D. M. & Miller, W. D. (2016), *The engineering design of systems: models and methods*, John Wiley & Sons, Hoboken, NJ, USA.

Burawoy, M. (2009), *The extended case method: Four countries, four decades, four great transformations, and one theoretical tradition*, University of California Press, Berkeley, CA, USA.

Caffall, D. S. & Michael, J. B. (2005), Formal methods in a system-of-systems development, *in* '2005 IEEE International Conference on Systems, Man and Cybernetics', Vol. 2, IEEE, pp. 1856–1863.

Carloni, L. P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A. L. et al. (2006), 'Languages and tools for hybrid systems design', *Foundations and Trends in Electronic Design Automation* **1**(1–2), 1–193.

Carroll, J. M. (1995), The scenario perspective on system development, *in* J. Caroll, ed., 'Scenario-

Based Design: Envisioning Work and Technology in System Development', Wiley, New York, NY, USA, pp. 1–17.

Casoto, P., Stevens, R. & Marco, G. D. (2014), 'Accident Response Use Case Engineering Analysis Report Using COMPASS Methods and Tools', EC COMPASS Project Deliverable D41.2; Online `http://www.compass-research.eu/Project/Deliverables/D41.2.pdf`. Accessed June 2016.

Cengarle, M. V., Bensalem, S., McDermid, J., Passerone, R., Sangiovanni-Vincentelli, A. & Törngren, M. (2013), 'Characteristics, capabilities, potential applications of cyber-physical systems: a preliminary analysis', EC CyPhERS Project Deliverable D2.1; Online `http://www.cyphers.eu/sites/default/files/D2.1.pdf`. Accessed June 2016.

Checkland, P. (1981), *Systems Thinking, Systems Practice*, Wiley, Chichester, UK.

Chiodo, M., Giusto, P., Jurecska, A., Hsieh, H. C., Sangiovanni-Vincentelli, A. & Lavagno, L. (1994), 'Hardware-software codesign of embedded systems', *IEEE micro* **14**(4), 26–36.

Clements, P. C. (1996), A survey of architecture description languages, *in* 'Proceedings of the 8th international workshop on software specification and design', IEEE Computer Society, p. 16.

Coleman, J. W., Lausdahl, K. & Larsen, P. G. (2014), Semantics of co-simulation, *in* J. Fitzgerald, P. G. Larsen & M. Verhoef, eds, 'Collaborative Design for Embedded Systems', Springer-Verlag, Berlin, Germany, pp. 273–292.

Collis, J. & Hussey, R. (2013), *Business research: A practical guide for undergraduate and postgraduate students*, Palgrave Macmillan, Basingstoke, UK.

Corbin, J. & Strauss, A. (2008), *Basics of qualitative research: Techniques and procedures for developing grounded theory*, Sage Publications, Thousand Oaks, CA, USA.

Coyle, R. G. (1996), *System dynamics modelling: a practical approach*, Chapman & Hall, London, UK.

Curran, J. & Blackburn, R. (2001), *Researching the small enterprise*, Sage Publications, London, UK.

Dabney, J. B. & Harman, T. L. (2004), *Mastering Simulink*, Prentice Hall, Upper Saddle River, NJ, USA.

Dahl, O.-J. & Nygaard, K. (1966), 'Simula: an algol-based simulation language', *Communications of the ACM* **9**(9), 671–678.

de Lama, N. & Sinclair, M. (2017), 'Recommendations for future research priorities, business opportunities and innovation strategies', EC Road2CPS Project Deliverable D4.1; Online `http://road2cps.eu/events/wp-content/uploads/2017/03/Road2CPS_644164_D4_1_Recommendations.pdf`. Accessed June 2016.

Deffuant, G., Amblard, F., Weisbuch, G. & Faure, T. (2002), 'How can extremism prevail? a study based on the relative agreement interaction model', *Journal of artificial societies and social simulation* **5**(4).

DeLaurentis, D. (2005), Understanding transportation as a system-of-systems design problem, *in* '43rd AIAA Aerospace Sciences Meeting and Exhibit', AIAA, New York, NY, USA.

Denzin, N. K. (1978), 'The research act: A theoretical orientation to sociological methods', *Journal of mixed methods research* **2**, 80–88.

Derler, P., Lee, E. A. & Vincentelli, A. S. (2012), 'Modeling cyber–physical systems', *Proceedings of the IEEE* **100**(1), 13–28.

Dodds, S. (2005), 'Designing improved healthcare processes using discrete event simulation', *The British Journal of Healthcare Computing & Information Management* **22**, 14–16.

Duindam, V., Macchelli, A., Stramigioli, S. & Bruyninck, H. (2009), *Modeling and control of complex physical systems: the port-Hamiltonian approach*, Springer-Verlag, Berlin, Germany.

Dzurec, L. C. & Abraham, I. L. (1993), 'The nature of inquiry: linking quantitative and qualitative research.', *Advances in Nursing Science* **16**(1), 73–79.

Easterby-Smith, M., Thorpe, R. & Jackson, P. R. (2012), *Management research*, Sage Publications, London, UK.

Eisenhardt, K. M. (1989), 'Building theories from case study research', *Academy of management review* **14**(4), 532–550.

Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S. & Xiong, Y. (2003), 'Taming heterogeneity – the Ptolemy approach', *Procedings of the IEEE* **91**(1), 127–144.

Engell, S., Paulen, R., Reniers, M. A., Sonntag, C. & Thompson, H. (2015), Core research and innovation areas in cyber-physical systems of systems, *in* M. R. Mousavi & C. Berger, eds, 'Cyber Physical Systems. Design, Modeling, and Evaluation', Springer International Publishing, Amsterdam, The Netherlands, pp. 40–55.

Engell, S., Paulen, R., Sonntag, C., Thompson, H., Reniers, M., Klessova, S. & Copigneaux, B. (2015), *Proposal of a European Research and Innovation Agenda on Cyber-physical Systems of Systems – 2016–2025*, Process Dynamics and Operations Group, TU Dortmund, Dortmund, Germany.

European Commission (2011), *Roadmap to a Single European Transport Area: Towards a Competitive and Resource Efficient Transport System (White Paper)*, Publications Office of the European Union.

Fagan, M. (1978), *A History of Engineering and Science in the Bell System: National Service in War and Peace*, Bell Telephone Laboratories, Murray Hill, NJ, USA.

Feiler, P. H., Gluch, D. P. & Hudak, J. J. (2006), The architecture analysis & design language (AADL): An introduction, Technical report, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, USA.

Ferris, T. L. (2009), On the methods of research for systems engineering, *in* 'Proceedings of the 7th Annual Conference on Systems Engineering Research', Loughborough University, Loughborough, UK.

Ferris, T. L., Cook, S. C. & Honour, E. C. (2005), 'Towards a structure for systems engineering research', *INCOSE International Symposium* **15**(1), 817–832.

Fiddy, E., Bright, J. & Hurrion, R. (1981), 'SEE-WHY: Interactive simulation on the screen', *Proceedings of the Institution of Mechanical Engineers* **293**, 167–172.

Fisher, D. (2006), An emergent perspective on interoperation in systems of systems, Technical report, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, USA.

Fishwick, P. A. & Zeigler, B. P. (1992), 'A multimodel methodology for qualitative model engineering', *ACM Transactions on Modeling and Computer Simulation (TOMACS)* **2**(1), 52–81.

Fitzgerald, J., Bryans, J. & Payne, R. (2012), A Formal Model-Based Approach to Engineering Systems-of-Systems, *in* L. Camarinha-Matos, L. Xum & H. Afsarmanesh, eds, 'Collaborative Networks in the Internet of Services, IFIP Advances in Information and Communication Technology', Springer, Berlin, Heidelberg, pp. 53–62.

Fitzgerald, J., Gamble, C., Larsen, P. G., Pierce, K. & Woodcock, J. (2015), Cyber-physical systems design: formal foundations, methods and integrated tool chains, *in* 'IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering (FormaliSE)', IEEE, pp. 40–46.

Fitzgerald, J., Larsen, P. G., Mukherjee, P., Plat, N. & Verhoef, M. (2005), *Validated Designs for Object–oriented Systems*, Springer, New York, NY, USA.

Fitzgerald, J., Larsen, P. G. & Verhoef, M. (2014*a*), From embedded to cyber-physical systems: Challenges and future directions, *in* J. Fitzgerald, P. G. Larsen & M. Verhoef, eds, 'Collaborative design for embedded systems', Springer-Verlag, Berlin, Germany, pp. 293–303.

Fitzgerald, J., Larsen, P. G. & Verhoef, M., eds (2014*b*), *Collaborative Design for Embedded Systems*, Springer-Verlag, Berlin, Germany.

Fitzgerald, J. & Pierce, K. (2014), Co-modelling and co-simulation in embedded systems design, *in* J. Fitzgerald, P. G. Larsen & M. Verhoef, eds, 'Collaborative Design for Embedded Systems', Springer-Verlag, Berlin, Germany, pp. 15–25.

Forcolin, M., Petrucco, P.-F., Previato, R., Stevens, R. L., Payne, R., Ingram, C. & Andrews, Z. (2013), 'Accident Response Use Case Engineering Analysis Report Using Current Methods and Tools', EC COMPASS Project Deliverable D41.1; Online `http://www.compass-research.eu/Project/Deliverables/D411.pdf`. Accessed June 2016.

Forrester, J. W. (1997), 'Industrial dynamics', *Journal of the Operational Research Society* **48**(10), 1037–1041.

Fowler, M. (2010), *Domain-specific languages*, Pearson Education, London, UK.

Fritzson, P. & Engelson, V. (1998), Modelica – A Unified Object-Oriented Language for System Modelling and Simulation, *in* 'Proceedings of the 12th European Conference on Object-Oriented Programming', Springer-Verlag, pp. 67–90.

Gagniuc, P. A. (2017), *Markov Chains: From Theory to Implementation and Experimentation*, John Wiley & Sons, Hoboken, NJ, USA.

Garlan, D., Monroe, R. & Wile, D. (1997), Acme: An architecture description interchange language, *in* 'Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research', pp. 7–22.

Geisberger, E. & Broy, M. (2012), *agendaCPS: Integrierte Forschungsagenda Cyber-Physical Systems*, Springer-Verlag, Berlin, Germany.

George, V. & Vaughn, R. (2003), 'Application of lightweight formal methods in requirement engineering', *CrossTalk: The Journal of Defense Software Engineering* **16**(1), 30–22.

Ghose, A. (2000), Formal methods for requirements engineering, *in* 'Proceedings of the International Symposium on Multimedia Software Engineering, 2000', IEEE, pp. 13–13.

Gill, H. (2008), 'From vision to reality: cyber-physical systems', *Presentation, HCSS National Workshop on New Research Directions for High Confidence Transportation CPS: Automotive, Aviation and Rail* .

Glaser, B. & Strauss, A. (1967), 'Grounded theory: The discovery of grounded theory', *Sociology The Journal Of The British Sociological Association* **12**, 27–49.

Graettinger, C. P., Garcia, S., Siviy, J., Schenk, R. J. & Van Syckle, P. J. (2002), Using the technology readiness levels scale to support technology management in the DoD's ATD/STO environments, Technical report, Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, USA.

Greasley, A. (1998), 'An example of a discrete-event simulation on a spreadsheet', *Simulation* **70**(3), 148–166.

Hafner-Zimmermann, S. & Henshaw, M. (2017), *The future of trans-Atlantic collaboration in modelling and simulation of Cyber-Physical Systems-A strategic research agenda for collaboration*, Steinbeis-Edition, Stuttgart, Germany.

Hardebolle, C. & Boulanger, F. (2009), 'Exploring multi-paradigm modeling techniques', *Simulation* **85**(11-12), 688–708.

Haren, V. (2011), *TOGAF Version 9.1*, Van Haren Publishing, Dordrecht, The Netherlands.

Harrell, C. R. & Lange, V. (2001), Healthcare simulation modeling and optimization using medmodel, *in* 'Proceedings of the 33nd conference on Winter simulation', IEEE Computer Society, pp. 233–238.

He, X., Ma, Z., Shao, W. & Li, G. (2007), A metamodel for the notation of graphical modeling languages, *in* 'Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International', Vol. 1, IEEE, pp. 219–224.

Heath, B. L. & Hill, R. R. (2010), 'Some insights into the emergence of agent-based modelling', *Journal of Simulation* **4**(3), 163–169.

Héder, M. (2017), 'From nasa to eu: the evolution of the trl scale in public sector innovation', *The Innovation Journal* **22**(2), 1–23.

Heemels, M. & Muller, G. (2007), *Boderc: Model-based design of high-tech systems*, 2nd edn, Embedded Systems Institute, Eindhoven, The Netherlands.

Hirshorn, S. R., Voss, L. D. & Bromley, L. K. (2017), *NASA Systems Engineering Handbook*, NASA, Washington DC, USA.

Hoare, C. A. R. (1978), 'Communicating sequential processes', *Communications of the ACM* **21**(8), 666–677.

Holt, J., Ingram, C., Larkham, A., Lloyd Stevens, R., Riddle, S. & Romanovsky, A. (2014), 'Convergence Report 3', EC COMPASS Project Deliverable D11.3; Online `http://www.compass-research.eu/Project/Deliverables/D11.3.pdf`. Accessed June 2016.

Holt, J. & Perry, S. (2008), *SysML for Systems Engineering*, Vol. 7, The Institution of Engineering and Technology, London, UK.

Holt, J. & Perry, S. (2010), *Modelling Enterprise Architectures*, The Institution of Engineering and Technology, London, UK.

Honour, E. (1999), Characteristics of engineering disciplines, *in* 'Proceedings of the 13th International Conference on Systems Engineering', Las Vegas, NV, USA.

Hurrion, R. D. (1976), The Design, use and Required Facilities of an Interactive Visual Computer Simulation Language to Explore Production Planning Problems, PhD thesis, University of London, London, UK.

Iliasov, A., Lopatkin, I. & Romanovsky, A. (2013), The safecap platform for modelling railway safety and capacity, *in* F. Bitsch, J. Guiochet & M. Kaâniche, eds, 'Computer Safety, Reliability, and Security', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 130–137.

INCOSE, International Council on Systems Engineering. (2007), International Council on Systems Engineering: Systems Engineering Vision 2020, Technical Report INCOSE-TP-2004-004-02, INCOSE, San Diego, CA, USA.

Ingram, C., Kristensen, K., Larkham, A., Larsen, P. G., Lloyd Stevens, R., Peleska, J., Riddle, S., Romanovsky, A. & Woodcock, J. (2013), 'Convergence Report 2', EC COMPASS Project Deliverable D11.2; Online `http://www.compass-research.eu/Project/Deliverables/D112.pdf`. Accessed June 2016.

ISO 16290, International Organization for Standardization. (2013), 'Space systems – Definition of the Technology Readiness Levels (TRLs) and their criteria of assessment'.

ISO/IEC 13817, International Organization for Standardization. (1996), 'Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method–Specification Language– Part 1: Base language'.

ISO/IEC/IEEE 15288, International Organization for Standardization. (2015), 'Systems and software engineering – System life cycle processes'.

ISO/IEC/IEEE 42010, International Organization for Standardization. (2011), 'Systems and software engineering – Architecture description'.

Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B. & Silva, O. (2004), Documenting component and connector views with uml 2.0, Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA.

Jaakkola, H. & Thalheim, B. (2010), 'Architecture-driven modelling methodologies', *Information Modelling and Knowledge Bases* **22**, 97–116.

Jahangirian, M., Eldabi, T., Naseer, A., Stergioulas, L. K. & Young, T. (2010), 'Simulation in manufacturing and business: A review', *European Journal of Operational Research* **203**(1), 1–13.

Jamshidi, M. (2009*a*), Introduction to System of Systems, *in* M. Jamshidi, ed., 'System of Systems Engineering: Innovations for the 21st Century', Wiley, New York, NY, USA, pp. 1–20.

Jamshidi, M. (2009*b*), *Systems of systems engineering: principles and applications*, CRC press, London, UK.

Jeffrey, P. & Seaton, R. (1995), 'The use of operational research tools: a survey of operational research practitioners in the UK', *Journal of the Operational Research Society* **46**(7), 797–808.

Jick, T. D. (1979), 'Mixing qualitative and quantitative methods: Triangulation in action', *Administrative science quarterly* **24**(4), 602–611.

Jones, C. B. (1990), *Systematic software development using VDM*, Vol. 2, Prentice-Hall, Englewood Cliffs, NJ, USA.

Karnopp, D. & Rosenberg, R. C. (1968), *Analysis and simulation of multiport systems*, MIT Press, Cambridge, MA, USA.

Karsai, G., Lang, A. & Neema, S. (2005), 'Design patterns for open tool integration', *Software & Systems Modeling* **4**(2), 157–170.

Keating, C., Rogers, R., Unal, R., Dryer, D., Sousa-Poza, A., Safford, R., Peterson, W. & Rabadi, G. (2003), 'System of systems engineering', *Engineering Management Journal* **15**(3), 36–45.

Kerlinger, F. (1986), *Foundations of Educational Research*, Holt, Rinehart & Wineston, New York, NY, USA.

Kervin, J. B. (1995), *Methods for business research*, Harper Collins, New York, NY, USA.

Kleijn, C. (2009), *20-sim 4.1 Reference Manual*, Controllab Products BV, Enschede, The Netherlands.

Kothari, C. R. (2004), *Research methodology: Methods and techniques*, New Age International, New Delhi, India.

Kotonya, G. & Sommerville, I. (1998), *Requirements engineering: processes and techniques*, Wiley Publishing, New York, NY, USA.

Kotov, V. (1997), Systems of Systems as Communicating Structures, Technical Report HPL-97-124, Computer Systems Laboratory, Hewlett Packard.

Kristensen, K., Lorenzen, L., Reese, B., Fill, T., Pedersen, S., Lauritsen, R. W., Nielsen, J. S. L. & Schulze, U. (2013), 'A/V/HA Ecosystems Prototype Using Current Methods and Tools', EC COMPASS Project Deliverable D42.1; Online `http://www.compass-research.eu/Project/Deliverables/D421.pdf`. Accessed June 2016.

Kristensen, K., Lorenzen, L., Reese, B., Fill, T., Pedersen, S., Lauritsen, R. W., Nielsen, J. S. L. & Schulze, U. (2014), 'A/V/HA Ecosystem Prototype Using Current Methods and Tools', EC COMPASS Project Deliverable D42.2; Online `http://www.compass-research.eu/Project/Deliverables/D42.2.pdf`. Accessed June 2016.

Kumar, R. (2014), *Research Methodology:A Step-by-Step Guide for Beginners*, Sage Publications, London, UK.

Kuuti, K. (1995), Work processes: scenarios as a preliminary vocabulary, *in* J. Caroll, ed., 'Scenario-Based Design: Envisioning Work and Technology in System Development', Wiley, New York, NY, USA, pp. 19–36.

Lankhorst, M. M., Proper, H. A. & Jonkers, H. (2009), 'The architecture of the archimate language', *Enterprise, Business-Process and Information Systems Modeling* **29**, 367–380.

Lapouchnian, A. (2006), Goal-oriented requirements engineering: An overview of the current research, *in* 'Depth Report for the Department of Computer Science', University of Toronto, Toronto, Canada.

Larsen, P. G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K. & Verhoef, M. (2010), 'The Overture Initiative Integrating Tools for VDM', *ACM SIGSOFT Software Engineering Notes* **35**(1), 1–6.

Larsen, P. G., Fitzgerald, J., Verhoef, M. & Pierce, K. (2014), Discrete-Event Modelling in VDM, *in* J. Fitzgerald, P. G. Larsen & M. Verhoef, eds, 'Collaborative Design for Embedded Systems', Springer-Verlag, Berlin, Germany, pp. 61–95.

Larsen, P. G., Gamble, C., Pierce, K., Ribeiro, A. & Lausdahl, K. (2014), Support for Co-modelling and Co-simulation: The Crescendo Tool, *in* J. Fitzgerald, P. G. Larsen & M. Verhoef, eds, 'Collaborative Design for Embedded Systems', Springer-Verlag, pp. 97–114.

Larsen, P. G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S. & Sahara, S. (2013), VDM-10 Language Manual, Technical Report TR-001, The Overture Initiative.

Lausdahl, K. G., Lintrup, H. K. A. & Larsen, P. (2008), Coupling Overture to MDA and UML, Master's thesis, Aarhus University/Engineering College of Aarhus.

Law, A. M., Kelton, W. D. & Kelton, W. D. (2000), *Simulation modeling and analysis*, 3rd edn, McGraw-Hill, New York, NY, USA.

Lee, E. A. (2003), Model-driven development – from object-oriented design to actor-oriented design, *in* 'Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation'.

Lee, E. A. (2008), Cyber physical systems: Design challenges, Technical Report UCB/EECS-2008-8, Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, USA.

Lee, E. A. (2010), CPS foundations, *in* 'Proceedings of the 47th Design Automation Conference', ACM, pp. 737–742.

Lee, E. A., Neuendorffer, S. & Wirthlin, M. J. (2003), 'Actor-oriented design of embedded hardware and software systems', *Journal of circuits, systems, and computers* **12**(03), 231–260.

Lee, E. A. & Seshia, S. A. (2011), *Introduction to embedded systems: A cyber-physical systems approach*, Lee & Seshia, Berkley, CA, USA.

Lee, E. A. & Zheng, H. (2005), Operational semantics of hybrid systems, *in* 'International Workshop on Hybrid Systems: Computation and Control', Springer, pp. 25–53.

Lee, N. & Lings, I. (2008), *Doing business research: a guide to theory and practice*, Sage, London, UK.

Leney, T., Coles, M., Grollman, P. & Vilu, R. (2004), *Scenarios toolkit*, Vol. 8, Office for Official Publications of the European Communities, Luxembourg City, Luxembourg.

Li, J., Pilkington, N. T., Xie, F. & Liu, Q. (2010), 'Embedded architecture description language', *Journal of Systems and Software* **83**(2), 235–252.

Liu, J. (1998), Continuous time and mixed-signal simulation in Ptolemy II, Technical Report UCB/ERL Memorandum M98/74, Deptartment of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, Berkeley, CA, USA.

Liu, L. & Yu, E. (2004), 'Designing information systems in social context: a goal and scenario modelling approach', *Information systems* **29**(2), 187–203.

Macal, C. M. & North, M. J. (2005), Validation of an agent-based model of deregulated electric power markets, *in* 'Procedings of North American Computational Social and Organization Science (NAACSOS) 2005', Wolters Kluwer, Alphen aan den Rijn, The Netherlands.

Macal, C. M. & North, M. J. (2010), 'Tutorial on agent-based modelling and simulation', *Journal of simulation* **4**(3), 151–162.

Magee, J., Dulay, N., Eisenbach, S. & Kramer, J. (1995), Specifying distributed software architectures, *in* 'European Software Engineering Conference', Springer, pp. 137–153.

Maier, M. W. (1998), 'Architecting principles for system-of-systems', *Systems Engineering* **1**(4), 267–284.

Maier, M. W. (2005), Research challenges for systems-of-systems, *in* 'IEEE International Conference on Systems, Man and Cybernetics', Vol. 4, IEEE, pp. 3149–3154.

Maler, O., Manna, Z. & Pnueli, A. (1991), From timed to hybrid systems, *in* J. W. de Bakker, C. Huizing,

W. P. de Roever & G. Rozenberg, eds, ‘Real-Time: Theory in Practice’, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 447–484.

Mankins, J. C. (1995), *Technology Readiness Levels, A White Paper*, NASA, Washington DC, USA.

Mankins, J. C. (2009), ‘Technology readiness assessments: A retrospective’, *Acta Astronautica* **65**(9-10), 1216–1223.

Martin, J. N. (1996), *Systems engineering guidebook: A process for developing systems and products*, Vol. 10, CRC Press, Boca Raton, FL, USA.

Mazanec, M. & Macek, O. (2012), On general-purpose textual modeling languages., *in* ‘Proceedings of the Dateso 2012 Workshop’, Vol. 12, pp. 1–12.

McKerchar, M. A. (2008), ‘Philosophical paradigms, inquiry strategies and knowledge claims: applying the principles of research design and conduct to taxation’, *EJournal of Tax Research* **6**(1), 5–22.

Medvidovic, N. & Taylor, R. N. (2000), ‘A classification and comparison framework for software architecture description languages’, *IEEE Transactions on software engineering* **26**(1), 70–93.

Midgley, G. (1997), Mixing methods: Developing systemic intervention, *in* J. Mingers & A. Gill, eds, ‘Multimethodology: The Theory and Practice of Combining Management Science Methodologies’, Wiley, Chichester, UK.

Minar, N., Burkhart, R., Langton, C., Askenazi, M. et al. (1996), The Swarm Simulation System: A toolkit for building multi-agent simulations, Technical report, Swarm Development Group, Santa Fe Institute, Santa Fe, NM, USA.

Mohammad, M. & Alagar, V. (2008), TADL-an architecture description language for trustworthy component-based systems, *in* ‘European Conference on Software Architecture’, Springer, pp. 290–297.

Morgan, G. & Smircich, L. (1980), ‘The case for qualitative research’, *Academy of management review* **5**(4), 491–500.

Mosterman, P. J. & Vangheluwe, H. (2004), ‘Computer automated multi-paradigm modeling: An introduction’, *Simulation* **80**(9), 433–450.

Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S. & Larsen, P. G. (2000), Exploring timing properties using VDM++, *in* ‘Proceedings of the second VDM workshop’.

Muller, G. (2013), 'Systems engineering research methods', *Procedia Computer Science* **16**, 1092–1101.

Myers, T., Dromey, G. & Fritzson, P. (2011), 'Comodeling: From requirements to an integrated software/hardware model', *Computer* **44**(4), 62–70.

Nicolescu, G., Boucheneb, H., Gheorghe, L. & Bouchhima, F. (2007), Methodology for efficient design of continuous/discrete-events co-simulation tools, *in* J. Anderson & R. Huntsinger, eds, 'High Level Simulation Languages and Applications', SCS Publishing, San Diego, CA, USA, pp. 172–179.

Nielsen, C. B., Larsen, P. G., Fitzgerald, J., Woodcock, J. & Peleska, J. (2015), 'Systems of systems engineering: basic concepts, model-based techniques, and research directions', *ACM Computing Surveys (CSUR)* **48**(2), 18.

Nuseibeh, B. & Easterbrook, S. (2000), Requirements engineering: a roadmap, *in* 'Proceedings of the Conference on the Future of Software Engineering', ACM, pp. 35–46.

OMG, Object Management Group. (2015*a*), 'Systems Modeling Language (SysML) 1.4', `http://www.omg.org/spec/SysML/1.4/PDF/`. Accessed June 2016.

OMG, Object Management Group. (2015*b*), 'Unified Modeling Language (UML) 2.5', `https://www.omg.org/spec/UML/2.5/PDF/`. Accessed June 2016.

Onwuegbuzie, A. J. & Leech, N. L. (2005), 'Taking the q out of research: Teaching research methodology courses without the divide between quantitative and qualitative paradigms', *Quality & Quantity* **39**(3), 267–295.

Pachl, J. (2009), *Railway Operation and Control*, 2nd edn, VTD Rail Publishing, Mountlake Terrace, WA, USA.

Pandey, R. (2010), 'Architectural description languages (ADLs) vs UML: a review', *ACM SIGSOFT Software Engineering Notes* **35**(3), 1–5.

Parker, J. & Epstein, J. M. (2011), 'A distributed platform for global-scale agent-based models of disease transmission', *ACM Transactions on Modeling and Computer Simulation (TOMACS)* **22**(1), 2.

Pérez-Martínez, J. E. & Sierra-Alonso, A. (2004), Uml 1.4 versus uml 2.0 as languages to describe software architectures, *in* 'European Workshop on Software Architecture', Springer, pp. 88–102.

Perry, D. E. & Wolf, A. L. (1992), ‘Foundations for the study of software architecture’, *ACM SIGSOFT Software engineering notes* **17**(4), 40–52.

Perry, S. & Holt, J. (2014), ‘Definition of the COMPASS Architectural Framework Framework’, EC COMPASS Project Deliverable D21.5b; Online `http://www.compass-research.eu/Project/Deliverables/D21.5b.pdf`. Accessed June 2016.

Pidd, M. (1992), Object orientation & three phase simulation, *in* ‘Proceedings of the 24th conference on Winter simulation’, ACM, pp. 689–693.

Pidd, M. (2004), *Computer simulation in management science*, 5th edn, Wiley, Chichester, UK.

Popper, S. W., Bankes, S. C., Callaway, R. & DeLaurentis, D. (2004), System of systems symposium: Report on a summer conversation, *in* ‘Proceedings of the 1st System of Systems Symposium’.

Potts, C. (1995), Using schematic scenarios to understand user needs, *in* ‘Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques’, ACM, pp. 247–256.

Rajkumar, R. R., Lee, I., Sha, L. & Stankovic, J. (2010), Cyber-physical systems: the next computing revolution, *in* ‘Proceedings of the 47th Design Automation Conference’, ACM, pp. 731–736.

Rechtin, E. (2000), *Systems architecting of organizations: Why eagles can’t swim*, CRC Press, Boca Raton, FL, USA.

Reghizzi, S. C., Breveglieri, L. & Morzenti, A. (2013), *Formal languages and compilation*, Springer-Verlag, New York, NY, USA.

Reniers, M. A. & Engell, S. (2014), ‘A european roadmap on cyber-physical systems of systems’, *ERCIM News* **97**.

Ridder, H.-G. (2017), ‘The theory contribution of case study research designs’, *Business Research* **10**(2), 281–305.

Robertson, D. A. & Caldart, A. A. (2009), *The dynamics of strategy: Mastering strategic landscapes of the firm*, Oxford University Press, Oxford, UK.

Robinson, S. (2014), *Simulation: the practice of model development and use*, 2nd edn, Palgrave Macmillan, Basingstoke, UK.

Robinson, S. & Higton, N. (1995), 'Computer simulation for quality and reliability engineering', *Quality and reliability engineering international* **11**(5), 371–377.

Rochard, B. P. & Schmid, F. (2000), 'A review of methods to measure and calculate train resistances', *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit* **214**(4), 185–199.

Rohrer, M. W. & McGregor, I. W. (2002), Automod: simulating reality using automod, *in* 'Proceedings of the 34th conference on Winter simulation: exploring new frontiers', Winter Simulation Conference, pp. 173–181.

RSSB, Rail Safety and Standards Board. (2000), 'Train Detection – Railway Group Standard GKRT0011 Iss 3'.

RSSB, Rail Safety and Standards Board. (2003), 'Interlocking Principles – Railway Group Standard GKRT0060 Iss 4'.

RSSB, Rail Safety and Standards Board. (2014), 'Lineside Signals, Indicators and Layout of Signals – Railway Group Standard GKRT0045 Iss 4'.

Rumbaugh, J., Jacobson, I. & Booch, G. (2004), *The Unified Modeling Language Reference Manual*, Pearson Higher Education, London, UK.

Ryan, M. J. (2013), An improved taxonomy for major needs and requirements artifacts, *in* 'INCOSE International Symposium IS2013', pp. 244–258.

Ryan, R., Scapens, R. W. & Theobald, M. (2002), *Research methods and methodology in accounting and finance*, Thomson Learning, London, UK.

Sanders, J. & Curran, E. (1994), *Software quality: a framework for success in software development and support*, Addison-Wesley, Wokingham, UK.

Schriber, T. J. (1974), *Simulation using GPSS*, Wiley, New York, NY, USA.

Seila, A. F. (2002), Spreadsheet simulation: spreadsheet simulation, *in* 'Proceedings of the 34th conference on Winter simulation: exploring new frontiers', Winter Simulation Conference, pp. 17–22.

Selic, B. (2003), 'The pragmatics of model-driven development', *IEEE software* **20**(5), 19–25.

Senge, P. M. (1990), *The art and practice of the learning organization*, Doubleday, New York, NY, USA.

Smith, J. D. (2005), An alternative to technology readiness levels for non-developmental item (ndi) software, *in* ‘Proceedings of the 38th Annual Hawaii International Conference on System Sciences’, IEEE, pp. 315–323.

Smith, J. K. (1983), ‘Quantitative versus qualitative research: An attempt to clarify the issue’, *Educational Researcher* **12**(3), 6–13.

Sommerville, I. (2007), *Software engineering*, Addison-Wesley, Harlow, UK.

Sommerville, I. & Sawyer, P. (1997), *Requirements engineering: a good practice guide*, John Wiley & Sons, Hoboken, NJ, USA.

Stake, R. E. (2005), Qualitative case studies, *in* N. K. Denzin & Y. S. Lincoln, eds, ‘The Sage handbook of qualitative research’, Sage Publications, Thousand Oaks, CA, USA, pp. 443–466.

Sterman, J. D. (2000), *Business dynamics: systems thinking and modeling for a complex world*, McGraw-Hill, Boston, MA, USA.

Stout, T. M. & Williams, T. J. (1995), ‘Pioneering work in the field of computer process control’, *Annals of the History of Computing, IEEE* **17**(1), 6–18.

Sztipanovits, J. & Karsai, G. (1997), ‘Model-integrated computing’, *Computer* **30**(4), 110–111.

Thomas, D. & Moorby, P. (2008), *The Verilog Hardware Description Language*, Springer Science & Business Media, New York, NY, USA.

Thompson, H. (2013*a*), Cyber-Physical Systems: Uplifting Europe’s Innovation Capacity, Technical report, European Commission Unit A3-DG CONNECT, Brussels, Belgium.

Thompson, H. (2013*b*), Systems of systems engineering and control, Technical report, European Commission Unit A3-DG CONNECT, Brussels, Belgium.

Thompson, H., Paulen, R., Reniers, M., Sonntag, C. & Engell, S. (2015), ‘Analysis of the state-of-the-art and future challenges in cyber-physical systems of systems’, EC CPSoS Project Deliverable D2.4; Online `http://www.cpsos.eu/wp-content/uploads/2015/02/D2-4-State-of-the-art-and-future-challenges-in-cyber-physical-systems-of-.pdf`. Accessed June 2016.

Toulson, R. & Wilmshurst, T. (2012), Embedded Systems, Microcontrollers and ARM, *in* R. Toulson &

T. Wilmshurst, eds, ‘Fast and Effective Embedded Systems Design’, Elsevier, Waltham, MA, USA, pp. 3–16.

Travis, J. & Kring, J. (2007), *LabVIEW for everyone: graphical programming made easy and fun*, Prentice-Hall, Upper Saddle River, NJ, USA.

US Department of Defence (2003), *Technology Readiness Assessment (TRA) Deskbook*, Deputy Under Secretary of Defense for Science and Technology, USA.

US Department of Defense (2003), *DoD Architecture Framework Version 1.0*, DoD Architecture Framework Working Group, USA.

Vahid, F. & Givargis, T. (2001), *Embedded System Design: A Unified Hardware/Software Introduction*, John Wiley & Sons, Hoboken, NJ, USA.

Valerdi, R., Axelband, E., Baehren, T., Boehm, B., Dorenbos, D., Jackson, S., Madni, A., Nadler, G., Robitaille, P. & Settles, S. (2008), ‘A research agenda for systems of systems architecting’, *International Journal of System of Systems Engineering* **1**(1-2), 171–188.

van Amerongen, J. (2010), *Dynamical systems for creative technology*, Controllab Products BV, Enschede, The Netherlands.

van Amerongen, J., Kleijn, C. & Gamble, C. (2014), Continuous-time modelling in 20-sim, *in* J. Fitzgerald, P. G. Larsen & M. Verhoef, eds, ‘Collaborative Design for Embedded Systems’, Springer-Verlag, Berlin, Germany, pp. 27–59.

Van Lamsweerde, A. (2003), From system goals to software architecture, *in* ‘International School on Formal Methods for the Design of Computer, Communication and Software Systems’, Springer, pp. 25–43.

Verhoef, M. (2009), Modeling and Validating Distributed Embedded Real-Time Control Systems, PhD thesis, Radboud University Nijmegen, The Netherlands.

Verhoef, M. & Larsen, P. G. (2007), Interpreting Distributed System Architectures Using VDM++ – A Case Study, *in* ‘5th Annual Conference on Systems Engineering Research’.

Verhoef, M., Larsen, P. G. & Hooman, J. (2006), Modeling and Validating Distributed Embedded Real-Time Systems with VDM++, *in* J. Misra, T. Nipkow & E. Sekerinski, eds, ‘FM 2006: Formal Methods’, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 147–162.

Verhoef, M., Pierce, K., Gamble, C. & Broenink, J. (2014), Collaborative development of embedded systems, *in* J. Fitzgerald, P. G. Larsen & M. Verhoef, eds, 'Collaborative Design for Embedded Systems', Springer-Verlag, Berlin, Germany, pp. 3–14.

Vogt, W. P. & Johnson, R. B. (2011), *Dictionary of Statistics & Methodology: A Nontechnical Guide for the Social Sciences*, Sage, London, UK.

Waite, M. & Hawker, S. (2009), *Oxford Paperback Dictionary & Thesaurus*, Oxford University Press, New York, NY, USA.

Walden, D. D., Roedler, G. J., Forsberg, K., Hamelin, R. D. & Shortell, T. M. (2015), *Systems engineering handbook: A guide for system life cycle processes and activities*, John Wiley & Sons, Hoboken, NJ, USA.

Walliman, N. (2011), *Your research project: Designing and planning your work*, Sage, London, UK.

Whitworth, I. R., Smith, S., Hone, G. & McLeod, I. (2006), How do we know that a scenario is "appropriate"?, *in* '11th International Command and Control Technology Symposium, Cambridge, UK'.

Winston, W. L. & Albright, S. C. (2015), *Practical management science*, Cengage Learning, Boston, MA, USA.

Wolf, W. (2007), 'The Good News and the Bad News (Embedded Computing Column)', *IEEE Computer* **40**(11), 104–105.

Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A. & Perry, S. (2012), Features of cml: A formal modelling language for systems of systems, *in* 'System of Systems Engineering (SoSE), 2012 7th International Conference on', IEEE, pp. 1–6.

Woodcock, J., Larsen, P. G., Bicarregui, J. & Fitzgerald, J. (2009), 'Formal methods: Practice and experience', *ACM computing surveys (CSUR)* **41**(4), 19.

Woods, E. & Hilliard, R. (2005), Architecture description languages in practice session report, *in* 'Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture', IEEE, pp. 243–246.

Yin, R. (2014), *Case Study Research: Design and Methods*, Sage Publications, Thousand Oaks, CA, USA.

Yu, E., Giorgini, P., Maiden, N. & Mylopoulos, J. (2011), *Social modeling for requirements engineering*, MIT Press, Cambridge, MA, USA.

Zachman, J. A. (2008), ‘Concise Definition of the Zachman Framework’, `https://www.zachman.com/about-the-zachman-framework`. Accessed June 2016.

Zeigler, B. P. & Zhang, L. (2015), Service-oriented model engineering and simulation for system of systems engineering, *in* ‘Concepts and Methodologies for Modeling and Simulation’, Springer, pp. 19–44.

Zhang, Z., Eyisi, E., Koutsoukos, X., Porter, J., Karsai, G. & Sztipanovits, J. (2014), ‘A co-simulation framework for design of time-triggered automotive cyber physical systems’, *Simulation Modelling Practice and Theory* **43**, 16–33.

Zhao, Y., Liu, J. & Lee, E. A. (2007), A programming model for time-synchronized distributed real-time systems, *in* ‘Real Time and Embedded Technology and Applications Symposium’, IEEE, pp. 259–268.

A CPSoS-AF Definition

A.1 Framework Definition

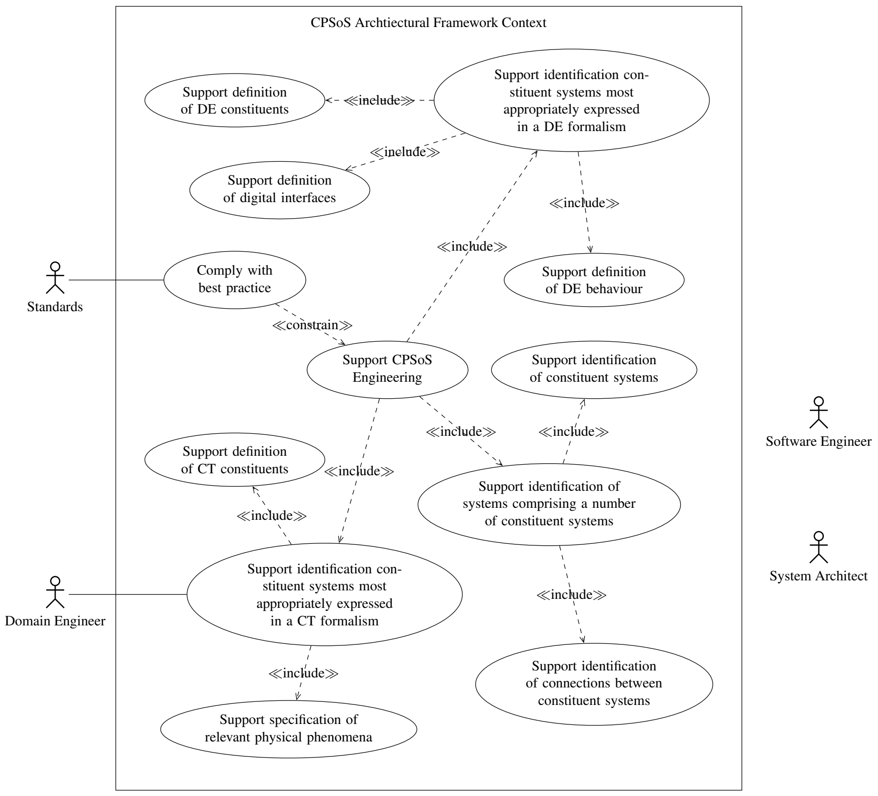

CPSoS Archtiectural Framework Context
Support definition of DE constituents
≪include≫
Support identification constituent systems most appropriately expressed in a DE formalism
≪include≫
Support definition of digital interfaces
≪include≫
≪include≫
Standards
Comply with best practice
Support definition of DE behaviour
≪constrain≫
Support CPSoS Engineering
Support identification of constituent systems
Software Engineer
≪include≫
≪include≫
Support definition of CT constituents
≪include≫
Support identification of systems comprising a number of constituent systems
≪include≫
System Architect
Domain Engineer
Support identification constituent systems most appropriately expressed in a CT formalism
≪include≫
≪include≫
Support identification of connections between constituent systems
Support specification of relevant physical phenomena

Figure A.1: Architectural Framework Context View

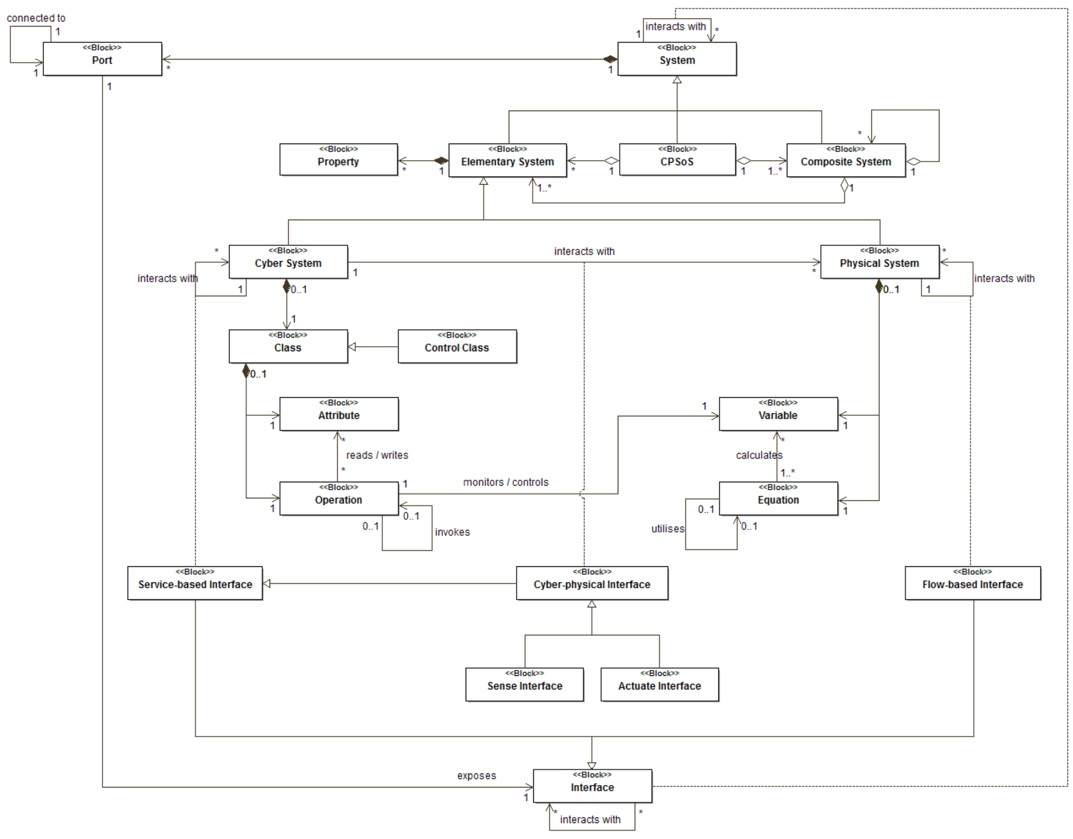
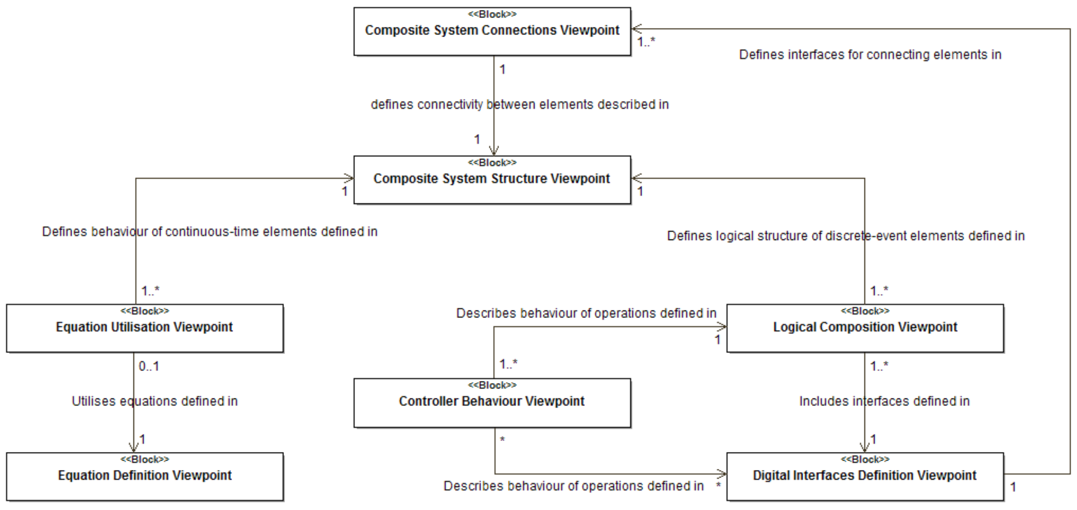


Figure A.2: Ontology Definition View



<<Block>>
Composite System Connections Viewpoint
1
defines connectivity between elements described in
1
<<Block>>
Composite System Structure Viewpoint
1..*
Defines interfaces for connecting elements in
1
Defines behaviour of continuous-time elements defined in
1
Defines logical structure of discrete-event elements defined in
1..*
<<Block>>
Equation Utilisation Viewpoint
0..1
Utilises equations defined in
1
<<Block>>
Equation Definition Viewpoint
Describes behaviour of operations defined in
1
1..*
<<Block>>
Logical Composition Viewpoint
1..*
1..*
<<Block>>
Controller Behaviour Viewpoint
Includes interfaces defined in
1
*
<<Block>>
Digital Interfaces Definition Viewpoint
Describes behaviour of operations defined in
*
1

Figure A.3: Viewpoint Relationships View

CPSoS-AF Rule 1: Each Composite System Structure View (CSSV) must have a corresponding Composite System Connections View (CSCV). Instances of each constituent must be defined and named in the CSCV, and any ports they have must be added.

CPSoS-AF Rule 2: For every *Elementary System* defined in a CSSV, it must be designated as being most appropriately modelled using a DE or CT formalism.

CPSoS-AF Rule 3: For every *Cyber System* defined in a CSSV, a Logical Composition View (LCV) must be defined.

CPSoS-AF Rule 4: For every *Operation* included in a LCV, a Controller Behaviour View (CBV) must be defined.

CPSoS-AF Rule 5: Every LCV must contain exactly one class which implements the *Controller Class* interface. The implementation of this controller interface is a special class which executes a main body periodically. The controller class must implement an operation named 'loop', containing the logic for top-level control of the cyber system.

CPSoS-AF Rule 6: For every *Physical System* defined in a CSSV, an Equation Utilisation View (EUV) must be defined.

CPSoS-AF Rule 7: Any equations used in an EUV must be defined in the Equation Definition Viewpoint (EDV).

CPSoS-AF Rule 8: In an effort to avoid ambiguity, ports defined in a CSCV to facilitate digital interaction must only be used in a single connection. Mulipliple connections utilising a single interface must be facilitated by additional ports.

CPSoS-AF Rule 9: All *Service-based Interfaces* exposed by a port must be defined in the Digital Interfaces Definition View (DIDV).

CPSoS-AF Rule 10: Where a service-based interface is used by a Cyber Constituent to monitor a Physical Constituent, its interface must implement the *Sensor Interface*. Any implementation of a sensor interface must contain a single attribute named ‘value’, which synchronises a single value with the CT environment. A single operation named ‘read’ should take no parameters, and return a single real number.

CPSoS-AF Rule 11: Where a service-based interface is used by a Cyber Constituent to control a Physical Constituent, its interface must implement the *Actuator Interface*. Any implementation of an actuate interface must contain a single attribute named ‘value’, which synchronises a single value with the CT environment. A single operation named ‘write’ should take a single real number as a parameter, and not offer a return type.

A.2 Viewpoint Definitions

A.2.1 Composite System Structure Viewpoint

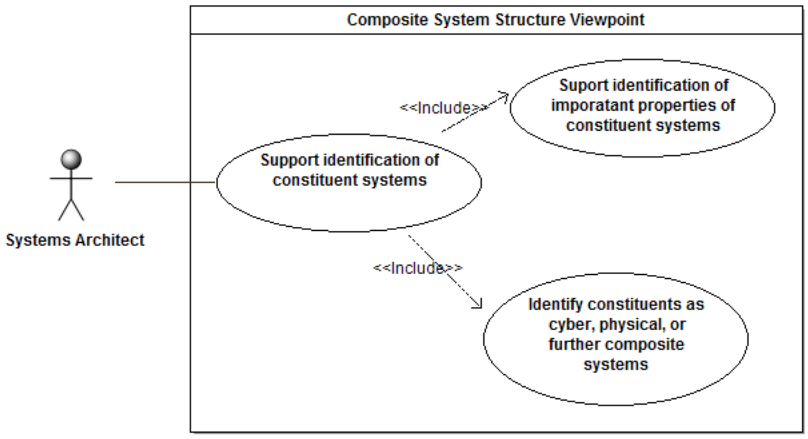

Composite System Structure Viewpoint
Systems Architect
Support identification of constituent systems
<<Include>>
Suport identification of imporatant properties of constituent systems
<<Include>>
Identify constituents as cyber, physical, or further composite systems

Figure A.4: Viewpoint Context View for Composite System Structure Viewpoint

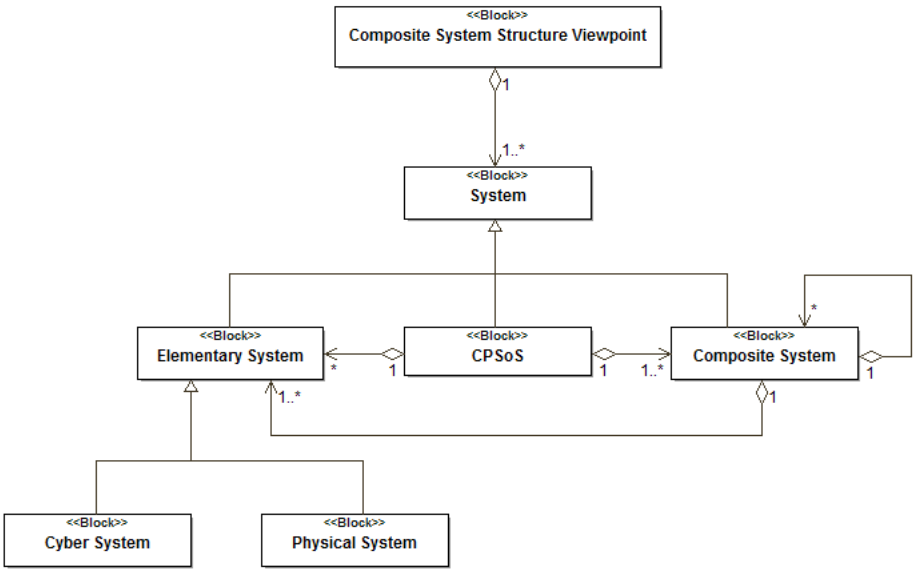


Figure A.5: Viewpoint Definition View for Composite System Structure Viewpoint

A.2.2 Composite System Connections Viewpoint

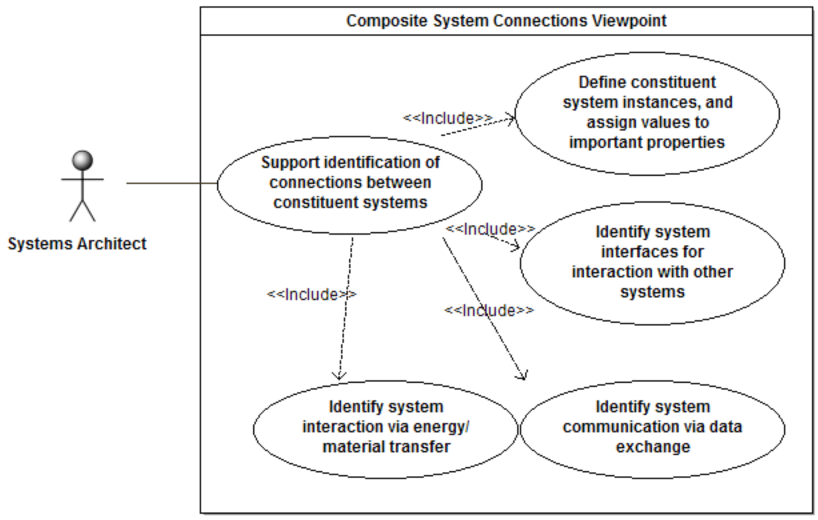

Composite System Connections Viewpoint
Systems Architect
Support identification of connections between constituent systems
Define constituent system instances, and assign values to important properties
Identify system interfaces for interaction with other systems
Identify system interaction via energy/ material transfer
Identify system communication via data exchange
<<Include>>
<<Include>>
<<Include>>
<<Include>>

Figure A.6: Viewpoint Context View for Composite System Connections Viewpoint

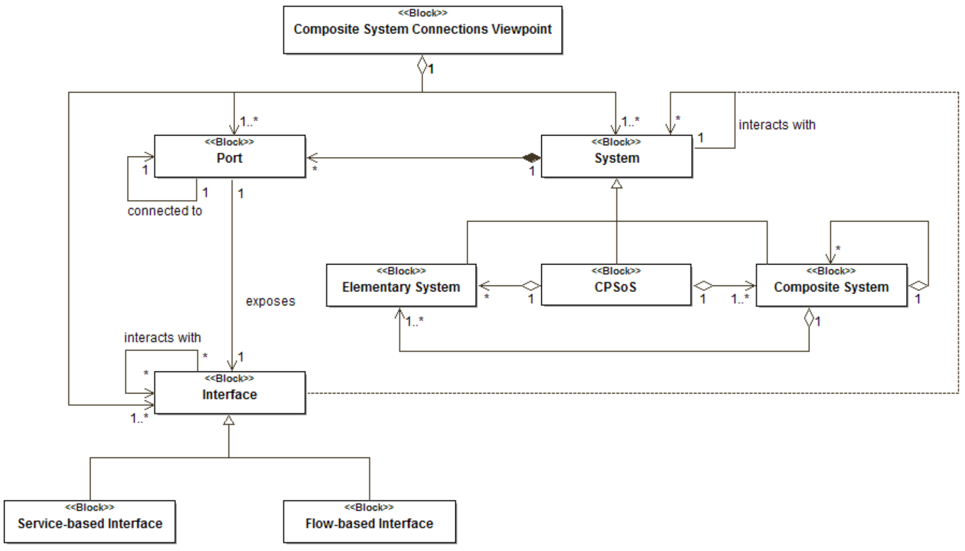

<<Block>> Composite System Connections Viewpoint
<<Block>> Port
<<Block>> System
<<Block>> Elementary System
<<Block>> CPSoS
<<Block>> Composite System
<<Block>> Interface
<<Block>> Service-based Interface
<<Block>> Flow-based Interface
interacts with
connected to
exposes
interacts with
1
1..*
1..*
*
1
1
1
1
1
*
1
1
1..*
*
1
*
1
1..*
*
*
1..*

Figure A.7: Viewpoint Definition View for Composite System Connections Viewpoint

A.2.3 Logical Composition Viewpoint

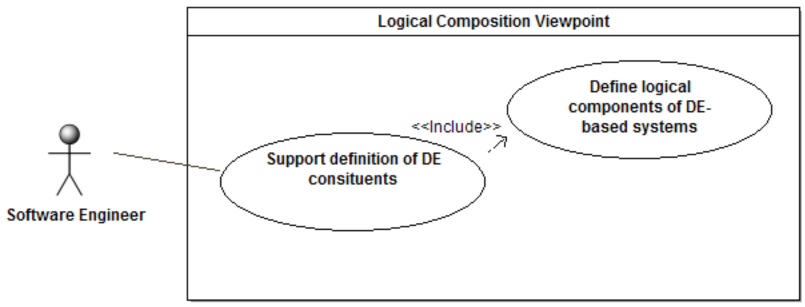

Logical Composition Viewpoint
Software Engineer
Support definition of DE consituents
<<Include>>
Define logical components of DE-based systems

Figure A.8: Viewpoint Context View for Logical Composition Viewpoint

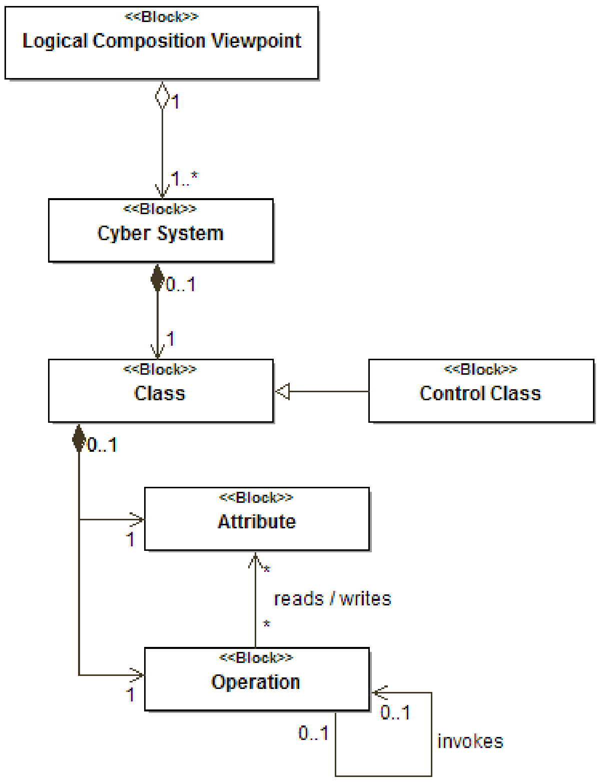


Figure A.9: Viewpoint Definition View for Logical Composition Viewpoint

A.2.4 Digital Interfaces Definition Viewpoint

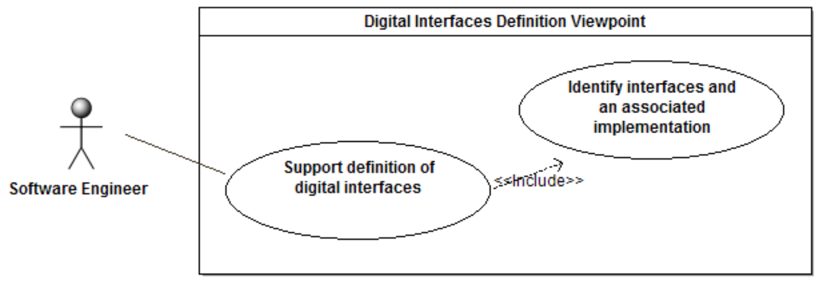

Digital Interfaces Definition Viewpoint
Software Engineer
Support definition of digital interfaces
Identify interfaces and an associated implementation
<<include>>

Figure A.10: Viewpoint Context View for Digital Interfaces Definition Viewpoint

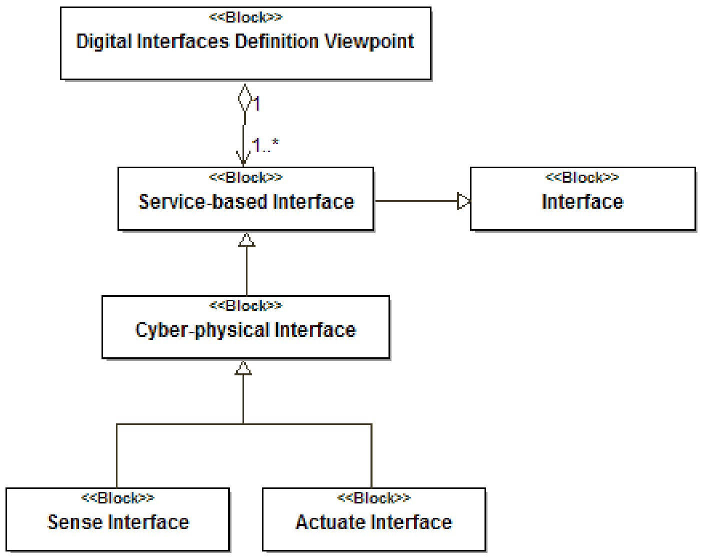


Figure A.11: Viewpoint Definition View for Digital Interfaces Definition Viewpoint

A.2.5 Controller Behaviour Viewpoint

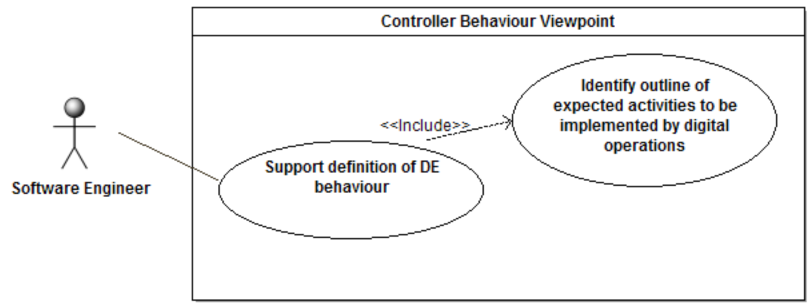

Controller Behaviour Viewpoint
Software Engineer
<<Include>>
Identify outline of expected activities to be implemented by digital operations
Support definition of DE behaviour

Figure A.12: Viewpoint Context View for Controller Behaviour Viewpoint

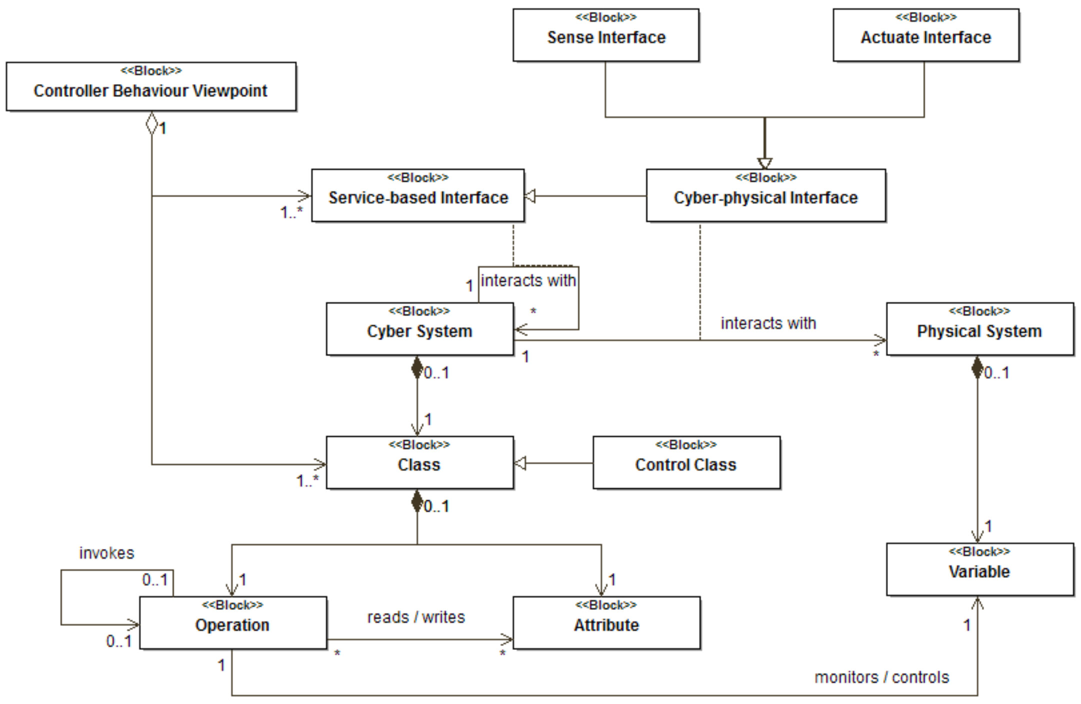

<<Block>> Sense Interface
<<Block>> Actuate Interface
<<Block>> Controller Behaviour Viewpoint
<<Block>> Service-based Interface
<<Block>> Cyber-physical Interface
interacts with
<<Block>> Cyber System
interacts with
<<Block>> Physical System
<<Block>> Class
<<Block>> Control Class
invokes
<<Block>> Operation
reads / writes
<<Block>> Attribute
<<Block>> Variable
monitors / controls

Figure A.13: Viewpoint Definition View for Controller Behaviour Viewpoint

A.2.6 Equation Definition Viewpoint

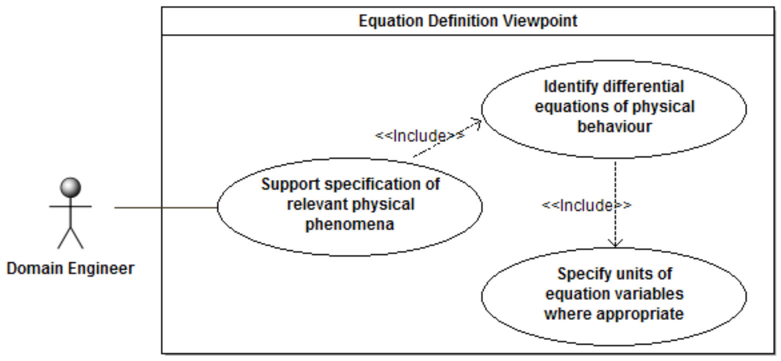

Equation Definition Viewpoint
Identify differential equations of physical behaviour
<<Include>>
Support specification of relevant physical phenomena
<<Include>>
Specify units of equation variables where appropriate
Domain Engineer

Figure A.14: Viewpoint Context View for Equation Definition Viewpoint

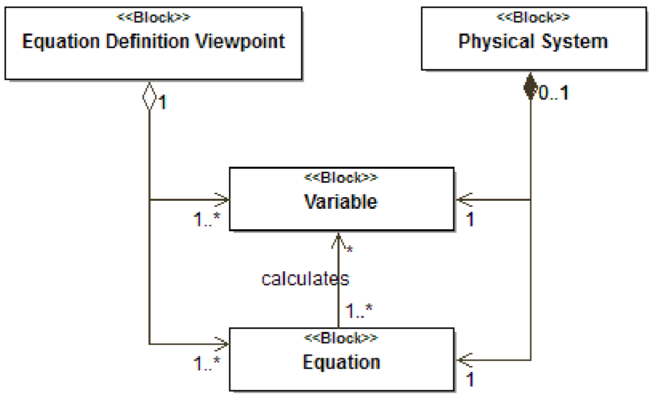


Figure A.15: Viewpoint Definition View for Equation Definition Viewpoint

A.2.7 Equation Utilisation Viewpoint

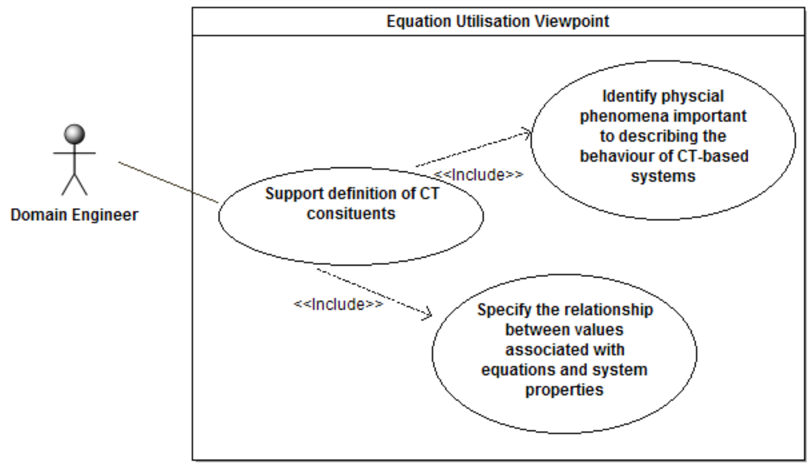

Equation Utilisation Viewpoint
Domain Engineer
Support definition of CT consituents
<<Include>>
Identify physcial phenomena important to describing the behaviour of CT-based systems
<<Include>>
Specify the relationship between values associated with equations and system properties

Figure A.16: Viewpoint Context View for Equation Utilisation Viewpoint

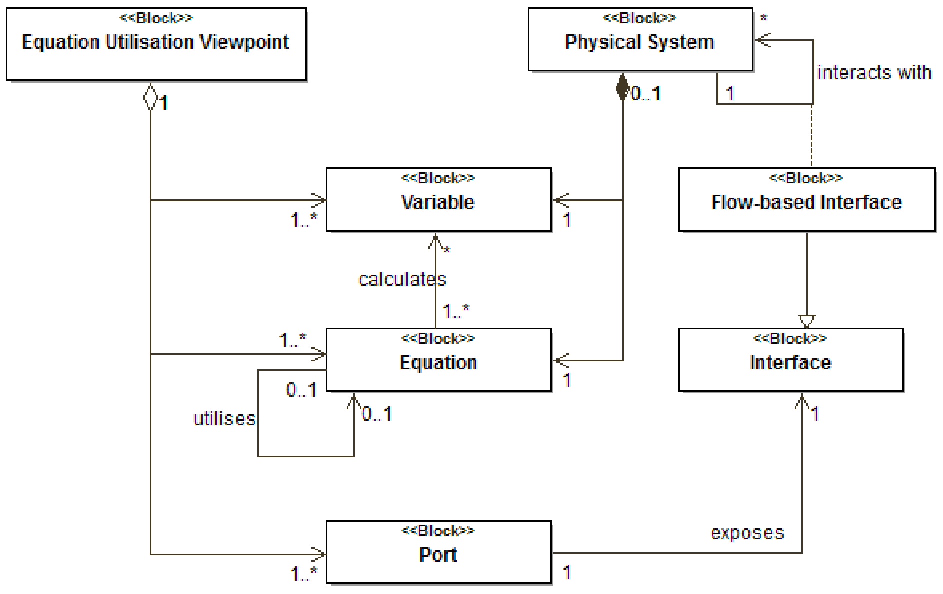


Figure A.17: Viewpoint Definition View for Equation Utilisation Viewpoint

B SysML Profile Definition

B.1 Composite Structures

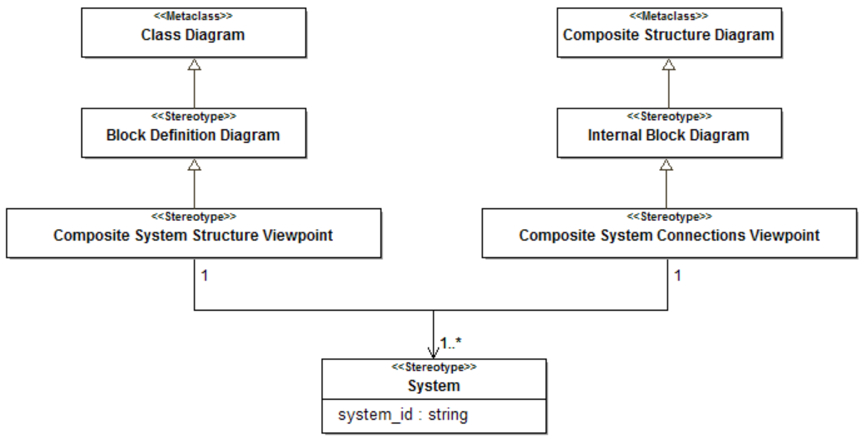


Figure B.1: Profile Diagram showing viewpoints describing composite structures

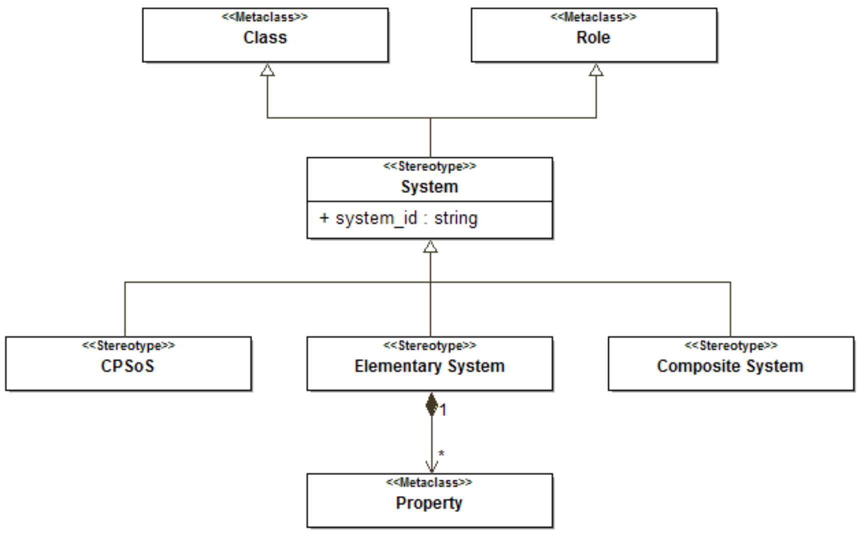
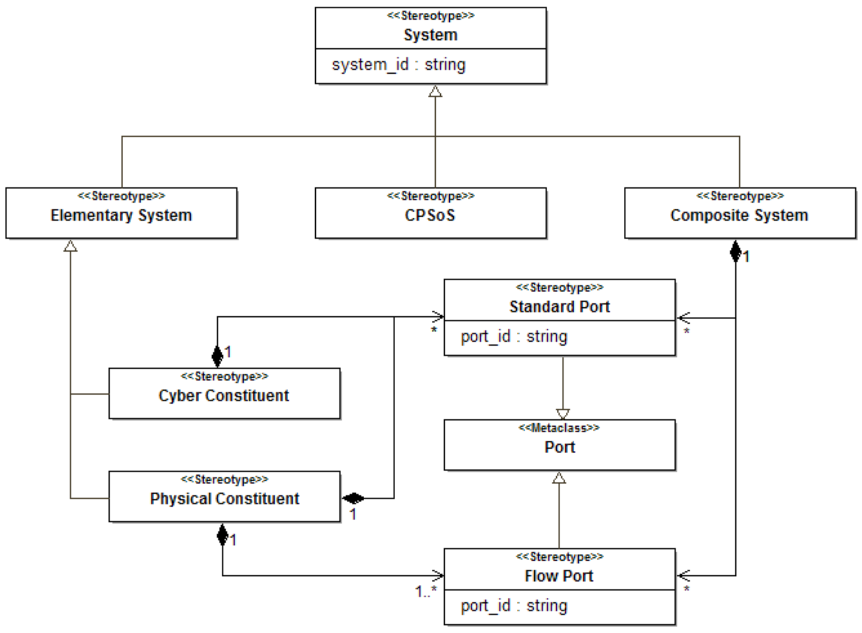


Figure B.2: Profile Diagram showing elements for describing composite structures



<<Stereotype>> System
system_id : string
<<Stereotype>> Elementary System
<<Stereotype>> CPSoS
<<Stereotype>> Composite System
1
<<Stereotype>> Standard Port
port_id : string
*
*
1
<<Stereotype>> Cyber Constituent
<<Metaclass>> Port
<<Stereotype>> Physical Constituent
1
1
<<Stereotype>> Flow Port
port_id : string
1..*
*

Figure B.3: Profile Diagram showing elements for describing composite connections

B.2 Physical Constituents

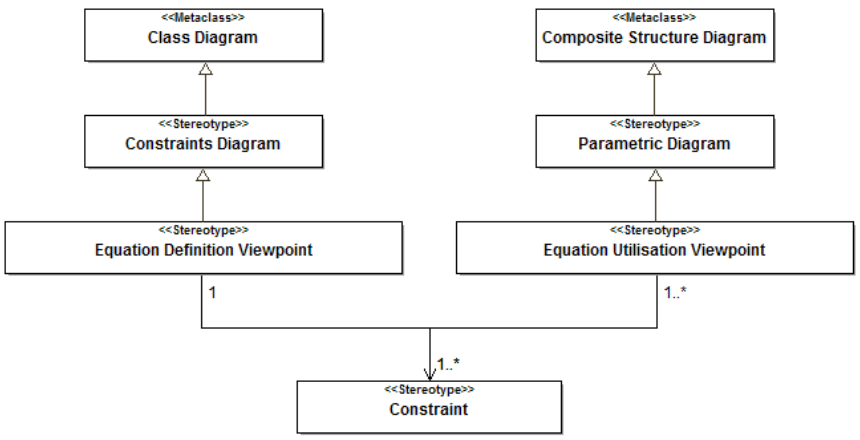


Figure B.4: Profile Diagram showing viewpoints describing physical constituents

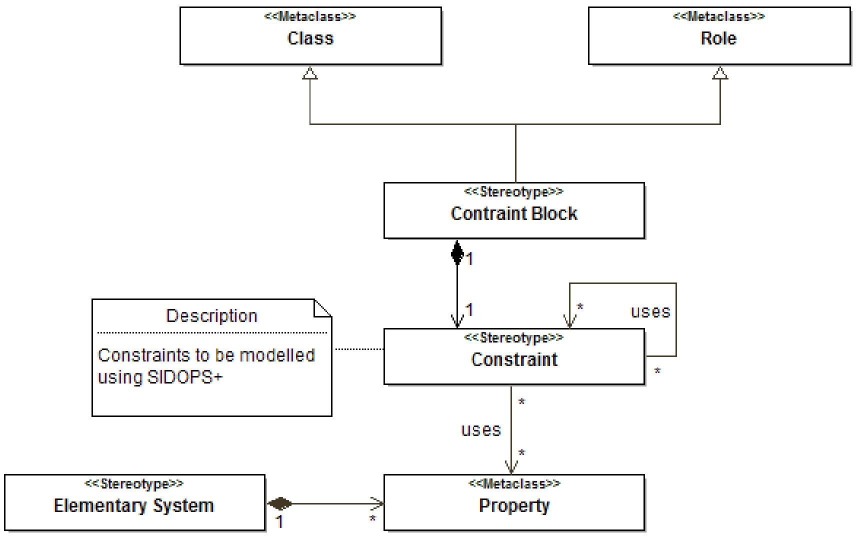


Figure B.5: Profile Diagram showing elements for describing CT behaviour

B.3 Cyber Constituents

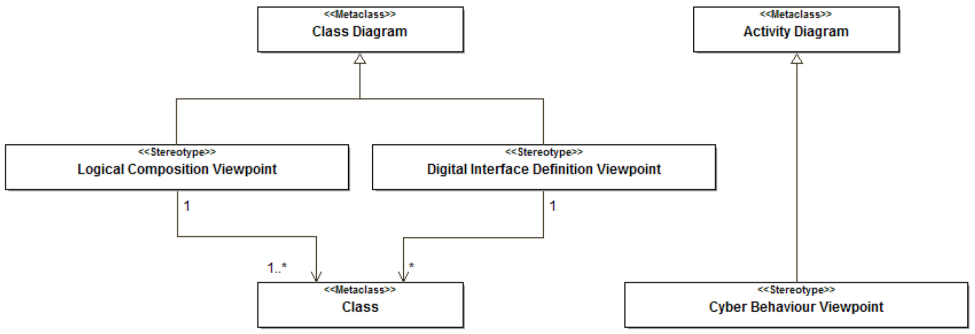


Figure B.6: Profile Diagram showing viewpoints describing cyber constituents

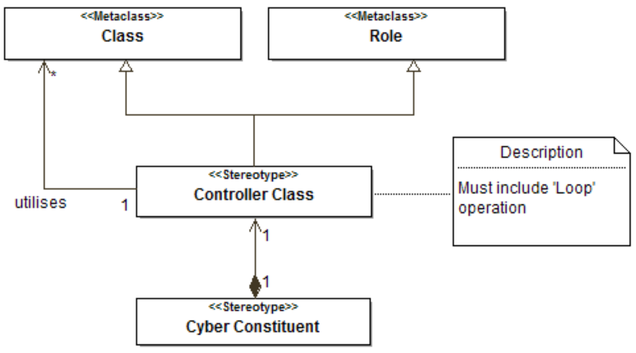


Figure B.7: Profile Diagram showing elements describing cyber structures

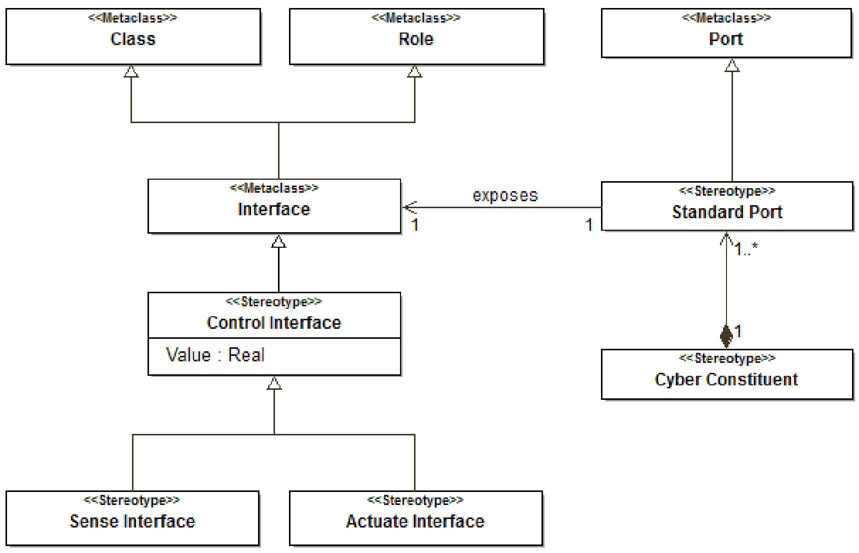


Figure B.8: Profile Diagram showing elements describing digital interfaces

C Abstract Syntax Definition

C.1 SysML Profile

$SysMLModel = CompositeContent$

$$\begin{aligned} CompositeContent :: \quad children &: Id \xrightarrow{m} Block \\ connections &: Flow\text{-}\mathbf{set} \end{aligned}$$

$$\begin{aligned} Block :: \quad ports &: Id \xrightarrow{m} PortType \\ contents &: CompositeContent \mid NodeDescription \end{aligned}$$

$NodeDescription = PhysicalNode \mid CyberNode$

$PortType = FlowPort \mid ServicePort$

$FlowPort :: direction : Direction$

$Direction = \text{IN} \mid \text{OUT} \mid \text{BOTH}$

$ServicePort :: exposes : Interface$

$Interface = CommunicationInterface \mid ControlInterface$

$$\begin{aligned} CommunicationInterface :: \quad interfaceName &: Id \\ parameters &: Id \xrightarrow{m} UnitType \\ returns &: [UnitType] \end{aligned}$$

$ControlInterface = SenseInterface \mid ActuateInterface$

$SenseInterface = CommunicationInterface$

where

$$inv\text{-}SenseInterface(\text{-}, p, r) \quad \triangleq \quad p = \{\,\} \land r = \text{REAL})$$

$$ActuateInterface = CommunicationInterface$$

where

$$inv\text{-}ActuateInterface(\text{-}, p, r) \quad \triangleq \quad \mathbf{let}\ x \in \mathbf{rng}\, p\ \ \mathbf{in}$$
$$\quad x = \text{REAL} \land \mathbf{card\,dom}\, p = 1 \land r = \mathbf{nil}$$

$$Flow = ItemFlow \mid InformationFlow$$

$$\begin{aligned} ItemFlow\ ::\ \ name &: Id \\ source &: PortReference \\ target &: PortReference \end{aligned}$$

$$\begin{aligned} InformationFlow\ ::\ \ name &: Id \\ implements &: Interface \\ providedBy &: PortReference \\ requiredBy &: PortReference \end{aligned}$$

$$\begin{aligned} PortReference\ ::\ owner &: [Id] \\ port &: Id \end{aligned}$$

$$\begin{aligned} PhysicalNode\ ::\ constraints &: Id \xrightarrow{m} Constraint \\ constants &: Id \xrightarrow{m} Constant \\ bindings &: Binding\text{-}\mathbf{set} \end{aligned}$$

$$\begin{aligned} Constraint\ ::\ \ body &: Equation \\ parameters &: Id \xrightarrow{m} UnitType \end{aligned}$$

$$\begin{aligned} Constant\ ::\ value &: ValueType \\ unit &: UnitType \end{aligned}$$

$$\begin{aligned} Binding\ ::\ \ id &: [Id] \\ participants &: ValueRef\text{-}\mathbf{set} \end{aligned}$$

$$ValueRef = ParameterRef \mid ConstantId \mid PortId$$

$$\begin{aligned} ParameterRef\ ::\ constraint &: Id \\ parameter &: Id \end{aligned}$$

$\mathit{ConstantId} = \mathit{Id}$

$\mathit{PortId} = \mathit{Id}$

$\mathit{CyberNode} = \mathit{Class}\text{-}\mathbf{set}$

$\mathit{Class} = \mathit{ControllerClass} \mid \mathit{AuxClass}$

$\mathit{ControllerClass} = \mathit{AuxClass}$

$$\begin{aligned} \mathit{AuxClass} :: \mathit{className} &: \mathit{Id} \\ \mathit{classBody} &: \ldots \end{aligned}$$

$\mathit{Id} = \mathbf{token}$

$\mathit{Equation} = \mathbf{token}$

$\mathit{UnitType} = \text{REAL} \mid \text{INTEGER} \mid \text{BOOLEAN}$

$\mathit{ValueType} = \mathbb{Z} \mid \mathbb{R} \mid \mathbb{B}$

C.2 Co-Models

$$\begin{aligned} \mathit{CoModel} :: \mathit{ct\text{-}model} &: \mathit{CTModel} \\ \mathit{de\text{-}model} &: \mathit{DEModel} \end{aligned}$$

C.2.1 20-sim Models

$\mathit{CTModel} = \mathit{CompositeModel}$

$$\begin{aligned} \mathit{CompositeModel} :: \mathit{elements} &: \mathit{Id} \xrightarrow{m} \mathit{Submodel} \\ \mathit{connections} &: \mathit{Connection}\text{-}\mathbf{set} \end{aligned}$$

$$\begin{aligned} \mathit{Submodel} :: \mathit{modelType} &: \mathit{CompositeModel} \mid \mathit{ElementaryModel} \\ \mathit{interface} &: \mathit{Id} \xrightarrow{m} \mathit{SubmodelPort} \end{aligned}$$

$$\begin{aligned} \mathit{SubmodelPort} :: \mathit{orientation} &: \mathit{Orientation} \\ \mathit{type} &: \text{SIGNAL} \mid \text{BOND} \end{aligned}$$

$Orientation = \text{INPUT} \mid \text{OUTPUT}$

$$\begin{aligned} Connection :: \; name &: [Id] \\ source &: PortReference \\ target &: PortReference \end{aligned}$$

$ElementaryModel = PhysicalModel \mid ControlLinkModel$

$$\begin{aligned} PhysicalModel :: \; valueMap &: Id \xrightarrow{m} ValueInfo \\ equations &: Equation\text{-}\mathbf{set} \end{aligned}$$

$ValueInfo = Parameter \mid Variable$

$$\begin{aligned} Parameter :: \; value &: valueType \\ unit &: UnitType \end{aligned}$$

$Variable :: \; unit : UnitType$

$Equation = isnotdefined$

$ControlLinkModel = Id \xrightarrow{m} External$

$External = \text{IMPORT} \mid \text{EXPORT}$

C.2.2 VDM-RT Models

$$\begin{aligned} DEModel :: \; system &: VDMSystem \\ contract &: Contract \end{aligned}$$

$$\begin{aligned} VDMSystem :: \; cpus &: Id \xrightarrow{m} CPU \\ busses &: Bus\text{-}\mathbf{set} \end{aligned}$$

$CPU :: \; controller : VDMController$

$$\begin{aligned} VDMController :: \; implementation &: Id \\ interfaces &: Id \xrightarrow{m} Id \end{aligned}$$

$Bus = Id\text{-}\mathbf{set}$

$Contract = Id \xrightarrow{m} SharedVariable$

$SharedVariable = \text{MONITORED} \mid \text{CONTROLLED}$

D Model Translation Definition

D.1 Structural Elements

$GenerateCoModel : SysMLModel \rightarrow CoModel$

$GenerateCoModel(sysml) \quad \triangleq$
$\quad mk\text{-}CoModel(GenerateCTModel(sysml), GenerateDEModel(sysml))$

$GenerateCTModel : CompositeContent \rightarrow CTModel$

$GenerateCTModel(root) \quad \triangleq \quad GenerateComposite(mk\text{-}Block(\{\,\}, root))$

$GenerateComposite : Block \rightarrow CompositeModel$

$GenerateComposite(node) \quad \triangleq \quad mk\text{-}CompositeModel($
$\quad \{id \mapsto GenerateSubmodel(node.contents.children(id), node.contents, id) \mid$
$\qquad id \in \textbf{dom}\, node.contents.children\},$
$\quad \{GenerateConnection(flow, node) \mid$
$\qquad flow \in DetermineCTConnections(node.contents.connections)\})$

```
$GenerateSubmodel : Block \times CompositeContent \times Id \to Submodel$

$GenerateSubmodel(node, env, ref) \quad \triangleq \quad mk\text{-}Submodel($
    **if** $is\text{-}CompositeContent(node.contents)$
    **then** $GenerateComposite(node)$
    **else** $GenerateElementary(node),$
    $\{id \mapsto GeneratePort(node.ports(id), env, mk\text{-}PortReference(ref, id)) \mid$
                                        $id \in DetermineCTPorts(node.ports)\})$
```

```
$DetermineCTPorts : Id \xrightarrow{m} PortType \to Id$-**set**

$DetermineCTPorts(ports) \quad \triangleq$
    $\{id \mid id \in$ **dom** $ports \cdot (is\text{-}FlowPort(ports(id)) \vee$
                        $(is\text{-}ServicePort(ports(id)) \wedge is\text{-}ControlInterface(ports(id).exposes)))\}$
```

```
$GeneratePort : PortType \times CompositeContent \times PortReference \to SubmodelPort$

$GeneratePort(port, env, ref) \quad \triangleq \quad mk\text{-}SubmodelPort($
    $DetermineOrientation(p, env, ref),$
    **if** $(is\text{-}ServicePort(port) \vee (is\text{-}FlowPort(port) \wedge port.direction \neq \text{BOTH}))$
    **then** SIGNAL
    **else** BOND)
```

```
$DetermineOrientation : PortType \times CompositeContent \times PortReference \to Orientation$

$DetermineOrientation(port, env, ref) \quad \triangleq$
    **if** $is\text{-}FlowPort(port)$
    **then cases** $port.direction$ **of**
            IN $\to$ INPUT
           OUT $\to$ OUTPUT
          BOTH $\to DetermineFlowDirection(port, env, ref)$
          **end**
    **else** $DetermineInterfaceOrienation(port, env, ref)$
```

$\mathit{DetermineFlowDirection} : \mathit{PortType} \times \mathit{CompositeContent} \times \mathit{PortReference} \to \mathit{Orientation}$

```
$\mathit{DetermineFlowDirection}(\mathit{port}, \mathit{env}, \mathit{ref}) \quad \triangleq$
    **let** $\mathit{itemFlows} = \{\mathit{flow} \mid \mathit{flow} \in \mathit{env.connections} \cdot \textit{is-ItemFlow}(\mathit{flow})\}$ **in**
    **if** $\exists \mathit{flow} \in \mathit{itemFlows} \cdot \mathit{flow.source} = \mathit{ref}$
    **then** OUTPUT
    **else if** $\exists \mathit{flow} \in \mathit{itemFlows} \cdot \mathit{flow.target} = \mathit{ref}$
        **then** INPUT
        **else** ...
```

$\mathit{DetermineFlowDirection} : \mathit{PortType} \times \mathit{CompositeContent} \times \mathit{PortReference} \to \mathit{Orientation}$

```
$\mathit{DetermineFlowDirection}(\mathit{port}, \mathit{env}, \mathit{ref}) \quad \triangleq$
    **let** $\mathit{itemFlows} = \{\mathit{flow} \mid \mathit{flow} \in \mathit{env.connections} \cdot \textit{is-ItemFlow}(\mathit{flow})\}$ **in**
    **if** $\mathit{ref.owner} \neq$ **nil**
    **then if** $\exists \mathit{flow} \in \mathit{itemFlows} \cdot \mathit{flow.source} = \mathit{ref}$
        **then** OUTPUT
        **else if** $\exists \mathit{flow} \in \mathit{itemFlows} \cdot \mathit{flow.target} = \mathit{ref}$
            **then** INPUT
            **else if** $\textit{is-CompositeContent}(\mathit{env.children}(\mathit{ref.owner}).\mathit{contents})$
                **then** $\mathit{DetermineFlowDirection}($
                    $p, \mathit{env.children}(\mathit{ref.owner}).\mathit{contents},$
                    $\textit{mk-PortReference}($**nil**$, \mathit{ref.port}))$
                **else** INPUT
    **else if** $\exists \mathit{flow} \in \mathit{itemFlows} \cdot \mathit{flow.source} = \mathit{ref}$
        **then** INPUT
        **else if** $\exists \mathit{flow} \in \mathit{itemFlows} \cdot \mathit{flow.target} = \mathit{ref}$
            **then** OUTPUT
            **else** INPUT
```

```
$DetermineInterfaceOrienation : PortType \times CompositeContent \times PortReference$
                              $\to Orientation$

$DetermineInterfaceOrienation(port, env, ref) \quad \triangleq$
    **let** $informationFlows = \{flow \mid flow \in env.connections \cdot is\text{-}InformationFlow(flow)\}$ **in**
    **if** $ref.owner \neq$ **nil**
    **then if** $\exists flow \in informationFlows \cdot flow.providedBy = ref$
        **then if** $is\text{-}SenseInterface(port)$
            **then** INPUT
            **else** OUTPUT
        **else if** $\exists flow \in informationFlows \cdot flow.requiredBy = ref$
            **then if** $is\text{-}ActuateInterface(port)$
                **then** INPUT
                **else** OUTPUT
            **else if** $is\text{-}CompositeContent(env.children(ref.owner).contents)$
                **then** $DetermineInterfaceOrienation($
                    $p, env.children(ref.owner).contents,$
                    $mk\text{-}PortReference($**nil**$, ref.port))$
                **else** INPUT
    **else if** $\exists flow \in informationFlows \cdot flow.providedBy = ref$
        **then if** $is\text{-}SenseInterface(port)$
            **then** OUTPUT
            **else** INPUT
        **else if** $\exists flow \in informationFlows \cdot flow.requiredBy = ref$
            **then if** $is\text{-}ActuateInterface(port)$
                **then** OUTPUT
                **else** INPUT
            **else** INPUT
```

```
$DetermineCTConnections : Flow$-**set** $\to Flow$-**set**

$DetermineCTConnections(flows) \quad \triangleq$
    $\{flow \mid flow \in flows \cdot (is\text{-}ItemFlow(flow) \lor$
                $(is\text{-}InformationFlow(flow) \land is\text{-}ControlInterface(flow.implements)))\}$
```

$GenerateConnection : Flow \rightarrow Connection$

$GenerateConnection(flow) \quad \triangleq$

$\quad mk\text{-}Connection(flow.name, DetermineSource(flow), DetermineTarget(flow))$

D.2 Continuous Time Equations

$GenerateElementary : NodeDescription \rightarrow ElementaryModel$

$GenerateElementary(node) \quad \triangleq$

$\quad$ **if** $is\text{-}PhysicalNode(node)$

$\quad$ **then** $GeneratePhysical(node)$

$\quad$ **else** $GenerateCyber(node)$

$GeneratePhysical : PhysicalNode \rightarrow ElementaryModel$

$GeneratePhysical(mk\text{-}PhysicalNode(constraints, constants, bindings)) \quad \triangleq$

$\quad mk\text{-}ElementaryModel($

$\quad \{id \mapsto GenerateParameter(constants(id)) \mid id \in \textbf{dom}\, constants\} \overset{m}{\cup}$

$\quad \{b.id \mapsto GenerateVariable(b.participants, constraints) \mid b \in bindings \cdot DetermineVariableBinding(b)\},$

$\quad \{BindVariables(constraint.body) \mid constraint \in \textbf{rng}\, constraints\})$

$GenerateParameter : Constant \rightarrow Parameter$

$GenerateParameter(c) \quad \triangleq \quad mk\text{-}Parameter(c.value, c.unit)$

$GenerateVariable : ParameterRef\textbf{-set} \times Id \xrightarrow{m} Constraint \rightarrow Variable$

$GenerateVariable(examples, env) \quad \triangleq \quad \textbf{let}\ e \in examples\ \textbf{in}$

$\quad mk\text{-}Variable(env(e.constraint).parameters(e.parameter))$

$DetermineVariableBinding : Binding\textbf{-set} \rightarrow \mathbb{B}$

$DetermineVariableBinding(b) \quad \triangleq \quad \forall ref \in b.participants \cdot is\text{-}ParameterRef(ref)$

$BindVariables : Equation \to Equation$

$BindVariables(e) \quad \triangleq \quad \ldots$

$GenerateCyber : Block \times Id \to ControlLinkModel$

$GenerateCyber(node, id) \quad \triangleq$

$\quad \textbf{let}\ ports = node.ports\ \ \textbf{in}$

$\quad \{GenerateExternalId(id, p) \mapsto DetermineExternalVariable(ports(p).exposes) \mid$

$\quad p \in \textbf{dom}\, ports \cdot is\text{-}ControlInterface(ports(p).exposes)\}$

D.3 Logical Composition

$GenerateDEModel : CompositeContent \to DEModel$

$GenerateDEModel(root) \quad \triangleq$

$\quad mk\text{-}DEModel(GenerateSystem(root), GenerateContract(root.children))$

$GenerateContract : Id \xrightarrow{m} Block \to Contract$

$GenerateContract(m) \quad \triangleq \quad DetermineSharedVariables(m) \overset{m}{\cup}$

$\quad \overset{m}{\bigcup}(\{GenerateContract(m(c).contents.children) \mid c \in \textbf{dom}\, m \cdot is\text{-}CompositeContent(m(c).contents)\})$

$DetermineSharedVariables : Id \xrightarrow{m} Block \to Id \xrightarrow{m} SharedVariable$

$DetermineSharedVariables(m) \quad \triangleq$

$\quad \overset{m}{\bigcup}(\{GenerateSharedVariables(node, m(node)) \mid node \in \textbf{dom}\, m \cdot DetermineCTLink(m(node))\})$

$DetermineCTLink : Block \to \mathbb{B}$

$DetermineCTLink(node) \quad \triangleq \quad \exists p \in \textbf{rng}\, node.ports \cdot is\text{-}ControlInterface(p.exposes);$

$GenerateSharedVariables : Block \to Id \xrightarrow{m} SharedVariable$

$GenerateSharedVariables(id, node) \quad \triangleq$

$\quad \{CombineIds(id, p) \mapsto GenerateSharedVariable(node.ports(p).exposes) \mid$

$\quad p \in \textbf{dom}\ node.ports \cdot is\text{-}ControlInterface(node.ports(p).exposes)\}$

$GenerateSharedVariable : ControlInterface \to SharedVariable$

$GenerateSharedVariable(i) \quad \triangleq \quad \textbf{if}\ is\text{-}SenseInterface(i)$

$\qquad \textbf{then}\ \text{MONITORED}$

$\qquad \textbf{else}\ \text{CONTROLLED}$

$GenerateSystem : CompositeContent \to System$

$GenerateSystem(root) \quad \triangleq \quad \textbf{let}\ controllers = GenerateControllers(root.children)\ \textbf{in}$

$\quad \{id \mapsto GenerateCPU(id, controllers(id)) \mid id \in \textbf{dom}\ controllers\},$

$\quad \{\{ResolveId(f.providedBy, root), ResolveId(f.requiredBy, root)\} \mid$

$\quad f \in root.connections \cdot DetermineControlInterface(f)\}$

$DetermineControlInterface : Flow \to \mathbb{B}$

$DetermineControlInterface(f) \quad \triangleq \quad \textbf{if}\ is\text{-}InformationFlow(f)$

$\qquad \textbf{then}\ is\text{-}ControlInterface(f)$

$\qquad \textbf{else false}$

$GenerateCPU : Id \times Block \to CPU$

$GenerateCPU(id, node) \quad \triangleq \quad \textbf{let}\ controller \in \{c \mid c \in node.contents \cdot is\text{-}ControllerClass(c)\}\ \textbf{in}$

$\quad mk\text{-}VDMController(controller.className, DetermineControllerInterfaces(id, node))$

$DetermineControllerInterfaces : Id \times Block \to Id \xrightarrow{m} Id$

$DetermineControllerInterfaces(id, node) \quad \triangleq$

$\quad \{id \curvearrowright \_ \curvearrowright p \mapsto node.ports(p).exposes.interfaceName \mid$

$\quad p \in \textbf{dom}\ node.ports \cdot is\text{-}ControlInterface(node.ports(p).exposes)\}$

$$GenerateControllers : Id \xrightarrow{m} Block \to Id \xrightarrow{m} Block$$

```
$GenerateControllers(tree) \quad \triangleq \quad DetermineCyberNodes(tree) \overset{m}{\cup}$
    $\overset{m}{\bigcup} (\{GenerateControllers(tree(id).contents.children) \mid$
    $id \in \textbf{dom}\, tree \cdot is\text{-}CompositeContent(tree(id).contents)\})$
```

$$DetermineCyberNodes : Id \xrightarrow{m} Block \to Id \xrightarrow{m} Block$$

```
$DetermineCyberNodes(m) \quad \triangleq$
    $\{id \mapsto m(id) \mid id \in \textbf{dom}\, m \cdot is\text{-}CyberNode(m(id).contents)\}$
```

$$ResolveId : PortReference \times CompositeContent \to Id$$

```
$ResolveId(p, env) \quad \triangleq$
    let $last = mk\text{-}PortReference(\textbf{nil}, p.port)$ in
    if $p.owner \neq \textbf{nil}$
    then let $node = env.children(p.owner).contents$ in
        if $is\text{-}CompositeContent(node)$
        then $ResolveId(last, node)$
        else $p.owner$
    else let $infoFlows = \{i \mid i \in env.connections \cdot is\text{-}InformationFlow(i)\}$ in
        let $flow \in infoFlows \cdot flow.providedBy = last \vee flow.requiredBy = last$ in
        if $flow.providedBy = last$
        then $ResolveId(last, env.children(flow.requiredBy.owner).contents)$
        else $ResolveId(last, env.children(flow.providedBy.owner).contents)$
```