Modelling System of Systems Interface Contract Behaviour

Oldrich Faldik[1], Richard Payne[2], John Fitzgerald[2], and Barbora Buhnova[3]
[1]Faculty of Business and Economics, Mendel University, Brno, Czech Republic
[2]School of Computing Science, Newcastle University, Newcastle upon Tyne, United Kingdom
[3]Faculty of Informatics, Masaryk University, Brno, Czech Republic
xfaldik@mendelu.cz, richard.payne@ncl.ac.uk, john.fitzgerald@ncl.ac.uk, buhnova@fi.muni.cz

A key challenge in System of Systems (SoS) engineering is the analysis and maintenance of global properties under SoS evolution, and the integration of new constituent elements. There is a need to model the constituent systems composing a SoS in order to allow the analysis of emergent behaviours at the SoS boundary. The Contract pattern allows the engineer to specify constrained behaviours to which constituent systems are required to conform in order to be a part of the SoS. However, the Contract pattern faces some limitations in terms of its accessibility and suitability for verifying contract compatibility.

To address these deficiencies, we propose the enrichment of the

Contract pattern, which hitherto has been defined using SysML and the COMPASS Modelling Language (CML), by utilising SysML and Object Constraint Language (OCL). In addition, we examine the potential of interface automata, a notation for improving loose coupling between interfaces of constituent systems defined according to the contract, as a means of enabling the verification of contract compatibility. The approach is demonstrated using a case study in audio/video content streaming.

1 Introduction

A System of Systems (SoS) is a collection of systems brought together for a task that none of the systems can accomplish on its own. Each constituent system (CS) keeps its own management, goals, and resources while coordinating within the SoS and adapting to meet SoS goals [14]. The independence of CSs within an SoS places challenges the description of the SoS architecture, and the verification of global behaviour.

There are several efforts to define architectural patterns to assist in the systematic description of SoS architectures. One such pattern is the 'Contract' pattern for specifying interfaces between CSs. A *contract* constrains the behaviour, in terms of operations and their ordering, in which a CS may engage as a 'good citizen' in an SoS. For an example of a contract see Figure 1 where one can find contract defining operations, values and invariants for the Leader Election Device discussed later in this paper. The composition of such contracts can be used to verify the behaviour of the SoS as a whole.

The Contract pattern is at the moment being realised using the SysML and the COMPASS Modelling Language (CML) [5]. The SoS architectural structure is given in SysML, augmented by contractual expressions given in CML. The latter is a formal modelling language for SoSs, combining the state-based VDM and process-based Circus languages. CML may be used to specify preconditions, postconditions and invariants of contracts [6]. SoS decriptions in SysML, with expression definitions in CML, may be completely converted to CML, permitting access to SoS analysis tools, such as simulation, model checking and theorem proving [4].

The previous work on the Contract Pattern has relied on the combination of SysML and CML notations. SysML is readily accessed by a range of engineering stakeholders, while the 'pure' CML is

J. Kofroň, J. Tumova (Eds.): Formal Engineering approaches
to Software Components and Architectures (FESCA'17)
EPTCS 245, 2017, pp. 1–15, doi:10.4204/EPTCS.245.1

© O. Faldik, R. Payne, J. Fitzgerald & B. Buhnova
This work is licensed under the
Creative Commons Attribution License.

intended for the specialist SoS engineer who knows the formal notation [12]. However, this approach suffers from two potential limitations. First, the use of CML may limit the ease of adoption of this approach by communities more familiar with model-based systems engineering approaches using SysML. Second, in order to complete the representation of SysML in CML, it is necessary to verify the compatibility of the contracts offered by interacting CSs; the current tools do not allow this to be done statically, and so approaches are limited to simulation (with a lack of exhaustivity) or model checking (limited by the CML model checker) [4].

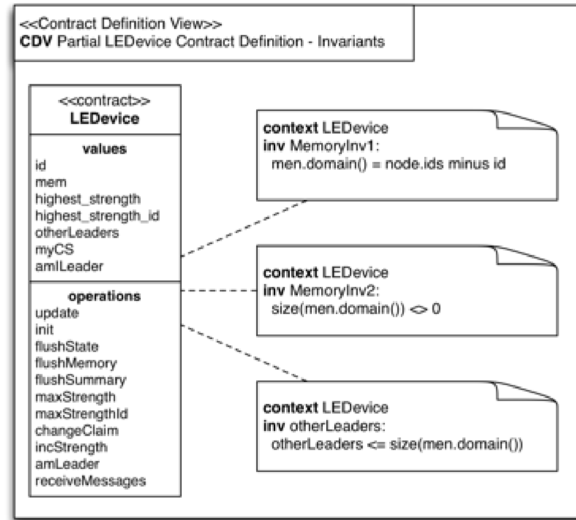

Figure 1: An example Contract Definition View with OCL Invariants

Paper Contribution: In this paper, we examine two approaches to the potential limitations in the use of the Contract pattern outlined above. First, we consider the potential of replacing CML by the better known Object Constraint Language (OCL) which is standardised by the Object Management Group (OMG) and is used to extend the Unified Modelling Language (UML). OCL may provide potential for wider take-up of the contractual approach, and may be seen as a more natural fit with SysML. Second, we examine the use of interface automata [1] as a way of increasing the opportunities for verification of contract conformance. This formalism was originally used for verification of the compatibility in component systems, and so may also be applicable in the SoS setting.

Paper Structure: We first describe the areas of related work (Section 2) and outline our approach, which utilises a case study in audio/video systems (Section 3). Section 4 describes the use of OCL for contract description. Section 5 discusses the application of interface automata in verifying the constitutent system compatibility. Section 6 applies interface automata to the case study, providing a basis for an initial evaluation, conclusions and future work (Section 7).

2 Related Work

In this section, we consider related work in the SoS domain, and in formal model-based methods.

Systems of Systems SoSs have had a considerable amount of research in both Europe and the US in recent years. Maier [15] characterises SoSs in terms of *operational* and *managerial independence*,

distribution, *evolution* and *emergent behaviour*. Maier, and later Dahmann [9], also categorise SoSs in terms of the levels of control over the CSs – from *directed SoSs* with some level of central control, to *virtual SoSs* with little to no central control. An output of the COMPASS project[1] was a detailed survey of SoS engineering concepts, model-based techniques and research directions [19].

One technique for tackling complexity and understanding the composition and connections between the CSs of an SoS is through architectural modelling. In [22], Payne and Fitzgerald survey several architecture description languages considering their applicability for modelling SoSs. SysML [21], an extended subset of UML, has found some traction in both academia and industry for modelling *systems* in terms of structure, behaviour and requirements.

SoSs pose many challenges to system engineers. Of those, the operational independence [15] of the CSs introduces complications when composing constituents to examine global behaviour. *Interface Contracts* [4] have been proposed as one method of dealing with the complexity – through detailing a collection of contracts which the CSs must conform. Arnold et al. have considered contracts in the context of SoS [3], using a combination of SysML/UPDM and a new language Goal and Contract Specification Language (GCSL), which combines OCL and Linear Temporal Logic (LTL). This work considers contracts at the SoS requirment level, rather than at the interfaces of individual CSs. In their survey paper [22], Payne and Fitzgerald evaluate architecture description languages and how contracts may be represented. The term Interface Contract has much in common with the Design by Contract [16] software engineering technique. An Interface Contract details the provided and required functionality of a CS, and dictates the state- and protocol-based requirements and guarantees of those functionalities.

In the SoS context, there is ongoing work in architectural patterns [13] which proposes an initial collection of topological patterns for SoS engineers, identifying aims, properties and risks. A *Contract Pattern* [4] has been defined which allows engineers to rigorously and consistently define the interface contracts of a SoS. This type of pattern considers a lower-level of abstraction to the aforementioned architectural pattern, and is considered to be an enabling pattern [24].

Formalisms There is a wide body of literature comparing formal methods and their industrial take-up [27]. In SoSs, the COMPASS Modelling Language (CML) is claimed to be the first formal language defined specifically for this domain [26]. CML is based on the languages VDM [10], CSP [11] and Circus [28]. A CML model is a collection of process definitions, each of which encapsulates a state and operations, and interacts with its environment via synchronous communications. The Contract pattern [4] proposes the use of CML to specify preconditions, postconditions and invariants of contract operations, as well as guards/actions on contract protocol state machines. The use of pre/postconditions is intended to allow CS interfaces to be specified in terms of ranges of permitted behaviour – important when each CS is independently owned and managed. The use of CML in this way allows an engineer to translate a profiled SysML model to a full CML model, and enables the verification afforded to CML [5].

There has been a large body of work integrating semi-formal notations such as UML and SysML with formal languages. In [2], the authors survey several efforts to provide a formal semantics to SysML – through translation to different formalisms. Included in their survey is a reference to the use of *interface automata*. Samir Chouali et al. [8] use an extended definition of interface automata to include pre and post conditions. A sequence of activities is performed to verify interface compatibility.

Whilst the above works apply formal notations to the verification of SysML models, the translation to these languages, or augmentation of diagrams with formal expressions, is often not a natural fit to SysML. By contrast, OCL [25] is a standardised language, defined by OMG, who also defined SysML.

[1] `www.compass-research.eu`

OCL allows expressions, preconditions, postconditions and other constraints to be defined *directly* in terms of the SysML modelling elements.

Contract pattern The Contract pattern [7] was introduced as a tool that guarantees pre-conditions and post-conditions of methods and invariants that constrain the state of objects. It means that, originally, it was designed for reliable classes.

The Contract pattern is a wide-spread programming approach to software designing which views the construction of software as based on contracts between clients and suppliers [17]. They rely on mutual commitments and benefits made explicitly expressed by statements. It has been developed in association with object-oriented programming. It is the basis for the programming language Eiffel and it is suitable for the design of component-based and agent systems [18].

Summary The contractual approach promoted in the Contract pattern has the potential to help in the integration of CSs and the verification of global behaviour. However, the pattern has weaknesses. The first relates to the use of CML in the expression defintion. Whilst useful for translation to CML for analysis, expressions stated in CML are not a natural fit with SysML. Second, CML does not support the analysis of contract compatibility well: simulation is not exhaustive, and the model checking capabilities are limited.

We propose an approach that firstly adopts OCL as the expression language in contract definitions. Secondly, we consider the use of *interface automata* for analysis of contract composition. While a mapping of SysML to interface automata has been demonstrated previously, this was limited to pure interface definitions and did not consider the contractual approach. In the next section we outline in more detail the main three areas that make up this approach.

3 Outline of the approach

In this section, we outline the notations forming the approach. First in Section 3.1, we detail the Contract Pattern and its views. Section 3.2 outlines OCL and its main characteristics, which we will use as an extension of the Contract Pattern instead of CML. In Section 3.3 we describe interface automata that will be later used for verification of contracts.

3.1 Contract pattern

As described in [23], there are many design patterns for SoS architecture. These can be categorised as *architectural patterns* and *enabling patterns*. The former describe specific system architecture patterns (such as a Centralised Architecture Pattern). The latter are specific constructs of modelling elements. The combination of these specific constructs and subsequent use enables many systems engineering applications. CSs may conform to multiple contracts and each may implement its contractual obligations in any way its owners choose. The Contract pattern [4] is an enabling pattern composed of several viewpoints (see Table 1) in SysML and CML.

Table 1: Informal description of the Contract Pattern viewpoints [4]

Name	**Purpose of View**
Contractual SoS Definition Viewpoint (CSDV)	Identifies the contracts which comprise the Contractual SoS.
Contract Conformance Viewpoint (CCV)	Identifies the constituent systems which make up the SoS and denotes the contracts to which those constituent systems conform. Includes all the contracts identified in the *CSDV*.
Contract Connections Viewpoint (CConnV)	Shows connections and interfaces between contracts of the Contractual SoS. Includes all the contracts identified in the *CSDV*.
Contract Definition Viewpoint (CDV)	Defines the operations, state variables and state invariants for a single contract identified in the *CSDV*.
Contract Protocol Viewpoint (CPV)	Defines the behaviour of a contract identified in the *CSDV* in terms of the ordering of messages between other members of the SoS and calls to the contract operations.

There is also the Interface pattern which is useful for defining data and interactions between CSs, but it is unsatisfactory in modelling the internal behaviour of CSs. The purpose of the Contract pattern is to enable specification of constraints on behaviours that each CS must deliver as an element of the SoS.

A limitation of the Contract pattern is that it does not precisely define which operation should and which should not be visible. In the case of the interface constituent system implementation this can lead to taking all operations as visible or as input actions. However, such an approach impairs the loose coupling of constituent system interfaces, manifesting itself by decreased flexibility of the entire system, e.g. by difficulties in replacing particular constituent systems due to other dependences. In order to solve this problem, an additional Interface definition view diagram from the Interface pattern is used, which does not improve clarity and transparency of the design because it is not stated in this contract-associated diagram.

3.2 Object Constraint Language (OCL)

OCL [20] is a language for describing constraints on a model. OCL expressions have formal semantics, and do not produce side effects influencing a described UML model [25].

OCL offers a compromise between a natural language description and strongly formal mathematical languages. Its typical application lies in the specification of invariants for classes and types, and definition of preconditions and postconditions. It allows the definition of elements for navigation of a SysML model, referring explicitly to specfic model elements, and their attributes. This allows OCL expressions to be checked for consistency with the underlying SysML model.

OCL is a strongly typed language that defines basic types and collections. It has a well-known, OMG standardized syntax, easing the application of the contract pattern for SoS stakeholders. Its use in this context is further described in Section 4.

3.3 Interface Automata

Formal description of component-based systems using *interface automata* was first introduced by Alfaro and Henzinger in 2001 [1]. This formal notation describes the interface of a component in a component system using *interface automaton* and allows verification of the component assembly.

Every *interface automaton* is composed of input actions that are modelled by methods exposed by the component to its environment, and thus they can be called. Input actions are designated by the symbol "?". Furthermore, there are output actions. These are methods required by the component from another component in a component system. Output actions are designated by the symbol "!". There are also internal actions that describe local methods of the component. Internal actions are designated by the symbol ";".

Definition 1. (Interface Automaton)
An *interface automaton* $A = \langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, \delta_A \rangle$ consists of

- S_A is a set of states.
- $I_A \subseteq S_A$ is a set of initial states.
- three disjoint sets Σ_A^I, Σ_A^O and Σ_A^H of inputs, outputs and hidden actions.
- $\delta_A \subseteq S_A \times \Sigma_A \times S_A$

The composition condition defines, that actions of two *interface automata* A_1 and A_2 are disjoint and asynchronous, except shared input and output actions. Shared actions are synchronized when A_1 and A_2 are composed. The following definition presents the composition condition.

Definition 2. (Composition)
The *interface automata* A_1, A_2 are composable if
$\Sigma_{A_1}^I \cap \Sigma_{A_2}^I = \Sigma_{A_1}^O \cap \Sigma_{A_2}^O = \Sigma_{A_1}^H \cap \Sigma_{A_2} = \Sigma_{A_1} \cap \Sigma_{A_2}^H = \emptyset$
If two *interface automata* A_1 and A_2 are composable then
$Shared(A_1,A_2) = (\Sigma_{A_1}^I \cap \Sigma_{A_2}^O) \cup (\Sigma_{A_2}^I \cap \Sigma_{A_1}^O)$

The synchronous product describes parallel execution of two *interface automata*.

Definition 3. (Synchronized product)
Let A_1, A_2 be two composable *interface automata*. The product $A_1 \otimes A_2$ is defined by

- $\Sigma_{A_1 \otimes A_2}^I = (\Sigma_{A_1}^I \cup \Sigma_{A_2}^I) \setminus Shared(A_1,A_2)$;
- $\Sigma_{A_1 \otimes A_2}^O = (\Sigma_{A_1}^O \cup \Sigma_{A_2}^O) \setminus Shared(A_1,A_2)$;
- $\Sigma_{A_1 \otimes A_2}^H = \Sigma_{A_1}^H \cup \Sigma_{A_2}^H \cup Shared(A_1,A_2)$;
- $((s_1,s_2),a,(s_1',s_2')) \in \delta_{A_1 \otimes A_2}$ if
  - $a \notin Shared(A_1,A_2) \wedge (s_1,a,s_1') \in \delta_{A_1} \wedge s_2 = s_2'$
  - $a \notin Shared(A_1,A_2) \wedge (s_2,a,s_2') \in \delta_{A_2} \wedge s_1 = s_1'$
  - $a \in Shared(A_1,A_2) \wedge (s_1,a,s_1') \in \delta_{A_1} \wedge (s_2,a,s_2') \in \delta_{A_2}$

Illegal states are states which are attainable by enabling internal actions or output actions.

Definition 4. (Illegal States)
Let A_1, A_2 be two composable *interface automata*. The set of illegal states $Illegal(A_1,A_2) \subseteq S_{A_1} \times S_{A_2}$ of $A_1 \otimes A_2$ is defined by $\{(s_1,s_2) \in S_{A_1} \times S_{A_2} | \exists a \in Shared(A_1,A_2).(a \in \Sigma_{A_1}^O(s_1) \wedge a \notin \Sigma_{A_2}^I(s_2)) \vee (a \in \Sigma_{A_2}^O(s_2) \wedge a \notin \Sigma_{A_1}^I(s_1))\}$.

The following algorithm describes verification of compatibility between A_1 and A_2. The result then either confirms or disproves the compatibility of components C_1 and C_2.

1. verify that A_1 and A_2 are composable,

2. calculate the product $A_1 \otimes A_2$,
3. calculate the set of illegal states in $A_1 \otimes A_2$,
4. calculate the bad states in $A_1 \otimes A_2$. The bad states represent states from which the illegal state are reachable by enabling only the internal action or the output actions (one suppose the existence of a helpful environment),
5. eliminate from the automaton $A_1 \otimes A_2$, the illegal state, the bad state, and the unreachable states from the initial states,
6. if the automaton $A_1 \otimes A_2$ is empty then the *interface automata* A_1, A_2 are not compatible, therefore C_1 and C_2 can not assembled correctly in any environment.

The complexity of this approach is linear on the size of A_1 and A_2 *interface automata*.

Interface automata in the Contract Pattern approach bring the possibility of verification of compatibility of contracts regarding operation visibility and complex analysis of the communication protocol of contracts.

We build upon the work of [4] in defining Interface Contracts using the Contract Pattern. However, as detailed in Section 2, the use of OCL may provide a more natural vehicle for combining a formal specification notation with the semi-formal SysML. Finally, we consider the use of *interface automata* to enrich these contracts and to model the abstract behaviour of CSs – complementing the state machines previously used in Interface Contracts. The use of *interface automata* provides a mechanism for verifying contract compatibility.

3.4 The Leader Election Case Study

In order to demonstrate the approach detailed in this paper, we use an industry-inspired case study [6]. The study presents an audio visual (AV) network of multiple AV devices with a network layer allowing communication between each device. Each device may be developed and maintained independantly, and this network exhibits the characteristic properties of SoSs. In the study each AV device must conform to a collection of contracts, each dictating required behaviours in order to be a part of the SoS.

Contracts exist for streaming of AV data, browsing digital content and for lower-level timing issues. This study has been previously defined with the Contract Pattern [4], which identifies three contracts: *Browsing Device*, *Streaming Device* and *LE Device*. As with that paper, we concentrate on the *LE Device*, in addition to the *Transport Layer* contract. In [4] the *LE Device* contract is defined using SysML diagrams and CML expressions to constrain various aspects of the contract definition.

4 OCL as an extension of SysML in CPS development

The idea of OCL applicability in SysML is based on the fact that SysML is defined as a UML profile[2]. This is achieved by using stereotypes and constraints applied to specific UML model elements.

OCL appears as applicable in the diagrams of Contract Definition Viewpoint and Contract Protocol Viewpoint that are basically similar to Class Diagram and State Machine Diagram. The remaining diagrams such as Contractual SoS Definition Viewpoint, Contract Conformance Viewpoint and Contract Connections Viewpoint are rather conceptual in nature and do not reach the required level of detail of description of the system where OCL should be applicable.

[2]A UML profile provides a mechanism for customising UML models for a particular domain or platform

4.1 Contract Definition Viewpoint

In [4], the Contract Definition Viewpoint (CDV) uses CML expressions to specify contract invariants, and the pre- and postconditions of contract operations. When applying OCL invariants and pre-/postconditions, we use SysML notes attached to *Contract* blocks with OCL expressions. Whilst some tools are able to analyse the conditions, we leave this for future work.

For the purposes of clarity, we split the definition of the LE Device contract across two diagrams, shown in Figures 1 and 2. The first diagram, Figure 1, presents the three invariants of the LE Device contract. The invariants are defined in separate SysML notes, each using the OCL expression notation for state variable invariants. The OCL statements define their *context* – here the LE Device contract – and then a named expression. In Figure 1, we see invariants which provide constraints which largely relate to the *mem* variable.

Figure 2 shows the constrained operations of the LE Device contract. It should be noted that there are several operations which do not require constraining. As above, we use SysML notes for each OCL statement – each relating to a separate operation. The OCL statement defines the context – here the LE Device contract, and the operation being constrained. For example, the top-most OCL statement refers to the *write* operation, giving the signature and then the precondition and postcondition for the operation. These conditions are OCL expressions.

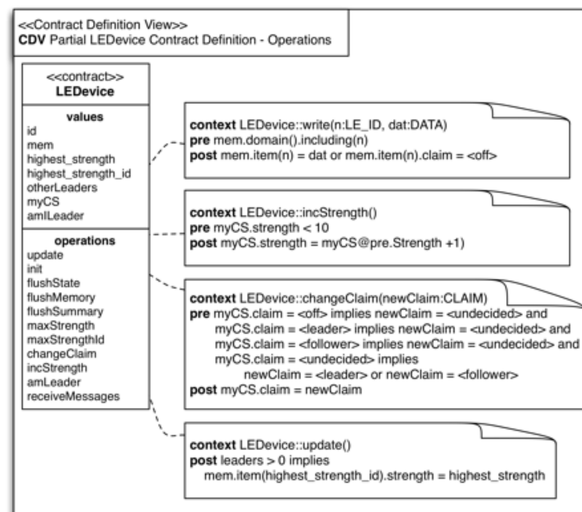

Figure 2: An example Contract Definition View with OCL Operations

4.2 Contract Protocol Definition Viewpoint

Finally, we consider the Contract Protocol Definition View. Previously, in this view, CML expressions are used to define guard constraints on transitions. We propose that these guards could be defined using OCL expressions. Figure 3 shows the LE Device behaviour with guarded transitions. In this example, the CML and OCL expressions are the same syntactically, and therefore are unchanged.

In this section, we have demonstrated the use of OCL in the context of the Contract Pattern. We feel that the use of OCL, rather than CML provides the advantage of accessibility to the SoS engineering architecture community.

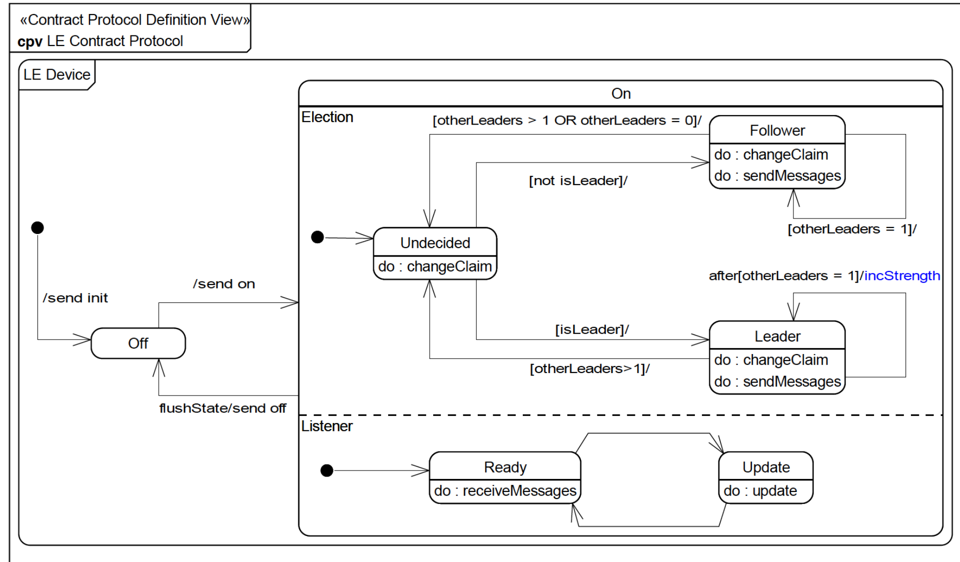

Figure 3: Example Contract Protocol Definition View

5 Verifying compatibility of contracts by interface automata

For verification of compatibility of contracts we present an application of *interface automata*, which takes into account the visibility of operations, which can exist as input, output or hidden operations. It also allows a complex view of the system using protocols for component communication. The design of *interface automata* for the contract in the present study is based on the design of *interface automata* described by Samir Chouali et al. [8] who extended the original definition by pre and post conditions, which also occur in the case of contracts. Furthermore, we extend this definition by formulation of variables, which can assume different values due to operation calls. Verification of compatibility of the two contracts is ensured by verification of their corresponding interfaces.

The following definition presents the *extended interface automaton* for contract.

Definition 5. (*Extended interface automaton* for contract)

Let be C a component of the component system CS, which represents a contractual specification for constituent system within a System of Systems. *Extended interface automata* associated with this component is nonuple

$A(C) = \langle S_A, I_A, \Sigma_A^I, \Sigma_A^O, \Sigma_A^H, V_A, Pre_A, Post_A, \delta_A \rangle$, which consists of

- S_A is the set of states.
- $I_A \subseteq S_A$ is a set of initial states.
- Σ_A^I, Σ_A^O and Σ_A^H are disjoint sets of inputs, outputs and hidden actions.
- V_A is the set of contract variables.
- Pre_A is the set of preconditions of component actions. Preconditions are specified in OCL.
- $Post_A$ is the set of postconditions of component actions. Postconditions are specified in OCL.
- δ_A is a plurality of transaction steps that are made on the basis of the occurrence of $a \in \Sigma$ actions in the state $s \in S$ under the valid precondition in Pre_A and postcondition in $Post_A$, which might include variables from V_A.

The following definition presents the composition of *interface automata* for contract.

Definition 6. (Composition of *extended interface automata* for contract)

Let $A_1 = \langle S_1, I_1, \Sigma_1^I, \Sigma_1^O, \Sigma_1^H, V_1, Pre_1, Post_1, \delta_1 \rangle$ and $A_2 = \langle S_2, I_2, \Sigma_2^I, \Sigma_2^O, \Sigma_2^H, V_2, Pre_2, Post_2, \delta_2 \rangle$ be two *extended interface automata*, and let's denote $\Sigma_1 = \Sigma_1^O \cup \Sigma_1^I \cup \Sigma_1^H$ and $\Sigma_2 = \Sigma_2^O \cup \Sigma_2^I \cup \Sigma_2^H$.

A_1 and A_2 are composable if $(\Sigma_1^I \cap \Sigma_2^I) = (\Sigma_1^O \cap \Sigma_2^O) = (\Sigma_1^H \cap \Sigma_2) = (\Sigma_1 \cap \Sigma_2^H) = \emptyset$.

If two *extended interface automata* A_1 and A_2 are composable then
$Shared(A_1, A_2) = (\Sigma_1^I \cap \Sigma_2^O) \cup (\Sigma_2^I \cap \Sigma_1^O)$.

The following definition presents the synchronized product of two *extended interface automata* for contract.

Definition 7. (Synchronized product of two *extended interface automata* for contract)

Let $A_1 = \langle S_1, I_1, \Sigma_1^I, \Sigma_1^O, \Sigma_1^H, V_1, Pre_1, Post_1, \delta_1 \rangle$ and $A_2 = \langle S_2, I_2, \Sigma_2^I, \Sigma_2^O, \Sigma_2^H, V_2, Pre_2, Post_2, \delta_2 \rangle$ are two composable *extended interface automata*.

The product $A_1 \otimes A_2$ is defined by $\langle S_1 \times S_2, I_1 \times I_2, \Sigma^I, \Sigma^O, \Sigma^H, V_1 \cup V_2, Pre, Post, \delta \rangle$ be such that:

- $\Sigma^I = (\Sigma_1^I \cup \Sigma_2^I) \backslash Shared(A_1, A_2)$;
- $\Sigma^O = (\Sigma_1^O \cup \Sigma_2^O) \backslash Shared(A_1, A_2)$;
- $\Sigma^H = \Sigma_1^H \cup \Sigma_2^H \cup Shared(A_1, A_2)$;
- $((s_1, s_2), pre, a, post, (s_1', s_2')) \in \delta$ if
  - $a \notin Shared(A_1, A_2) \wedge (s_1, pre_1, a, post_1, s_1') \in \delta_1 \wedge s_2 = s_2' \wedge pre = pre_1 \wedge post = post_1$
  - $a \notin Shared(A_1, A_2) \wedge (s_2, pre_2, a, post_2, s_2') \in \delta_2 \wedge s_1 = s_1' \wedge pre = pre_2 \wedge post = post_2$
  - $a \in Shared(A_1, A_2) \wedge ((s_1, pre_1, a, post_1, s_1') \in \delta_1 \wedge a \in \Sigma^I) \wedge ((s_2, pre_2, a, post_2, s_2') \in \delta_2 \wedge a \in \Sigma^O) \wedge pre = (pre_2 \wedge pre_1) \wedge post = (post_1 \wedge post_2)$
  - $a \in Shared(A_1, A_2) \wedge ((s_1, pre_1, a, post_1, s_1') \in \delta_1 \wedge a \in \Sigma^O) \wedge ((s_2, pre_2, a, post_2, s_2') \in \delta_2 \wedge a \in \Sigma^I) \wedge pre = (pre_1 \wedge pre_2) \wedge post = (post_2 \wedge post_1)$
- $Pre = Pre_1 \cup Pre_2 \cup \{(pre_1 \wedge pre_2) | pre_1 \in Pre_1 \wedge pre_2 \in Pre_2\}$;
- $Post = Post_1 \cup Post_2 \cup \{(post_1 \wedge post_2) | post_1 \in Post_1 \wedge post_2 \in Post_2\}$.

The following definition presents the illegal states of two *extended interface automata* for contract. A set of illegal states contains states in which shared actions between *extended interface automata* are not synchronized (because required functionality by one of the automata is not provided by the other), or no transition is enabled due to the restrictions resulting from the preconditions and postconditions of the enabled transitions from the state.

Definition 8. (Illegal states of two *extended interface automata* for contract)

Let A_1, A_2 be two composable *interface automata*. The set of illegal states $Illegal(A_1, A_2) \subseteq S_{A_1} \times S_{A_2}$ of $A_1 \otimes A_2$ is defined by $\{(s_1, s_2) \in S_{A_1} \times S_{A_2} | \exists a \in Shared(A_1, A_2).(a \in \Sigma_{A_1}^O(s_1) \wedge a \notin \Sigma_{A_2}^I(s_2)) \vee (a \in \Sigma_{A_2}^O(s_2) \wedge a \notin \Sigma_{A_1}^I(s_1))\} \cup \{(s_1, s_2) \in S_{A_1} \times S_{A_2} | \forall ((s_1, s_2), pre, a, post, (s_1', s_2')) \in \delta_{A_1 \otimes A_2}.((pre \equiv false) \vee (post \equiv false))\}$.

6 Translating the Leader Election Case Study to Extended Interface Automata

In this section, we will apply the above mentioned formalism to the Leader Election study. In order to convert the contract into an *extended interface automaton*, it is necessary to use the Contract Definition View which describes preconditions, values and operations (input, output, hidden), and the Contract Protocol Definition View to identify states and transitions between them. The following code presents the *extended interface automata* for *LE device* and *Transport Layer*.

LE device

- $S_{LD} = \{Off, OnFollower, OnLeader, OnUndecided, OnReady, OnUpdate\}$ 1
- $I_{LD} = \{Off\}$ 2
- $\Sigma^I_{LD} = \{receiveMessages\}$ 3
- $\Sigma^O_{LD} = \{sendMessages\}$ 4
- $\Sigma^H_{LD} = \{changeClaim, flushState, update, maxStrength, maxStrengthId,$ 5
  $incStrength, init, flushMemory, flushSummary, isLeader, write, turnOn, turnOff\}$ 6
- $V_{LD} = \{id, mem, highest_strength, highest_strength_id, otherLeaders, myCS, isLeader\}$ 7
- $Pre_{LD} = \{LDPreCC, LDPreW, LDPreIS\}$ where: { 8
  context *LE Device::changeClaimm(newClaim : Claim)* 9
  pre LDPreCC: *myCS.c* $=< off > \implies$ *newc* $=< undecided >$ 10
  and myCS.c = $< undecided > \implies$ *(newc* = $< leader >$ *or newc* = $< follower >$*)* 11
  and myCS.c = $< leader > \implies$ *newc* = $< undecided >$ 12
  and myCS.c = $< follower > \implies$ *newc* = $< undecided >$ 13
  14
  context *LE Device::write(n: LE_Id, dat: DATA)* ***pre LDPreW:*** *n in set dom mem* 15
  context *LE Device::incStrength()* ***pre LDPreIS:*** *myCS.s* $<$ *10* 16
  } 17
- $Post_{LD} = \{LDPostCC, LDPostW, LDPostIS\}$ where: { 18
  context *LE Device::changeClaimm(newClaim : Claim)* ***post LDPostCC:*** *myCS.c = newClaim* 19
  context *LE Device::write(n: LE_Id, dat: DATA)* ***post LDPostW:*** *mem(n) = dat or mem(n).c* = $< off >$ 20
  context *LE Device::incStrength()* ***post LDPostIS:*** *myCS.s = myCS .s + 1* 21
  } 22
- $\delta_{LD} = \{$ 23
  - Off : ***turnOn*** : $OnReady$ 24
  - $OnReady$: ***receiveMessages*** : $OnUpdate$ 25
  - $OnUpdate$: ***update*** : $OnReady$ 26
  - $OnReady$: ***turnOff*** : Off 27
  - Off : ***turnOn*** : $OnUndecided$ 28
  - $OnUndecided$: $LDPreCC$: ***changeClaim*** : $LDPostCC$: $OnFollower$ 29
  - $OnFollower$: $LDPreCC$: ***changeClaim*** : $LDPostCC$: $OnUndecided$ 30
  - $OnUndecided$: $LDPreCC$: ***changeClaim*** : $LDPostCC$: $OnLeader$ 31
  - $OnLeader$: $LDPreCC$: ***changeClaim*** : $LDPostCC$: $OnUndecided$ 32
  - $OnFollower$: ***sendMessages*** : $OnFollower$ 33
  - $OnLeader$: ***sendMessages*** : $OnLeader$ 34
  - $OnUndecided$: ***turnOff*** : Off 35
  - $OnFollower$: ***turnOff*** : Off 36
  - $OnLeader$: ***turnOff*** : Off 37

  } 38
  39

Transport Layer

- $S_{TL} = \{Init, Ready, CreateMessage, AddtoQueue, GetMessage,$ 40
  $CreateUnreachableMessage, TurnDeviceOn, TurnDeviceOff, SendtoDevice, ReceivedMessage\}$ 41

- $I_{TL} = \{Init\}$ 42
- $\Sigma^I_{TL} = \{sendMessages\}$ 43
- $\Sigma^O_{TL} = \{receiveMessages\}$ 44
- $\Sigma^H_{TL} = \{init, addToQueue, getNextMsg, createMessage,$ 45
  $AddToQueue, setDeviceOn, setDeviceOff, ready\}$ 46
- $V_{TL} = \{queue, devOn\}$ 47
- $Pre_{TL} = \{TLPreGNM, TLPreSDOF, TLPreSDON\}$ where: { 48
  context *Transport Layer::getNextMsg()* ***pre TLPreGNM:*** *queue → notEmpty* 49
  context *Transport Layer::setDeviceOff(in devId : LE_Id)* ***pre TLPreSDOF:*** *devOn[devId]→notEmpty* 50
  context *Transport Layer::setDeviceOn(in devId : LE_Id)* ***pre TLPreSDON:*** *devOn[devId]→notEmpty* 51
  } 52
- $Post_{TL} = \{TLPostI, TLPostATQ\}$ where: { 53
  context *context TransportLayer::Init()* ***post TLPostI:*** *devOn.domain() = node_ids and* 54
  devOn.range = false and queue.size() = 0 55
  56
  context *context TransportLayer::addToQueue(m:MSG)* ***post TLPostATQ:*** 57
  queue.size() = queue@pre.size() + 1 and queue.lastItem() = 58
  m and queue@pre = queue(1,...,queue.size()) 59
  } 60
- $\delta_{TL} = \{$ 61
  - $Init$: ***init*** : $TLPostI$: $ready$ 62
  - $Ready$: ***sendMessages*** : $ReceivedMessage$ 63
  - $ReceivedMessage$: ***createMessage*** : $CreateMessage$ 64
  - $CreateMessage$: ***addToQueue*** : $TLPostATQ$: $AddtoQueue$ 65
  - $AddtoQueue$: ***ready*** : $Ready$ 66
  - $Ready$: $TLPreGNM$: ***getNextMsg*** : $GetMessage$ 67
  - $GetMessage$: ***receiveMessages*** : $SendtoDevice$ 68
  - $SendtoDevice$: ***ready*** : $Ready$ 69
  - $GetMessage$: ***createMessage*** : $CreateUnreachableMessage$ 70
  - $SendtoDevice$: ***createMessage*** : $CreateUnreachableMessage$ 71
  - $CreateUnreachableMessage$: ***addToQueue*** : $AddToQueue$ 72
  - $AddToQueue$: ***ready*** : $Ready$ 73
  - $Ready$: $TLPreSDON$: ***setDeviceOn*** : $TurnDeviceOn$ 74
  - $TurnDeviceOn$: ***ready*** : $Ready$ 75
  - $Ready$: $TLPreSDOF$: ***setDeviceOff*** : $TurnDeviceOff$ 76
  - $TurnDeviceOff$: ***ready*** : $Ready$ 77

  } 78
  79

On lines 3-6 and 43-46, the classification of operations based on whether they are input, output or hidden deserves careful attention. The application of OCL for defining precondition and postcondition are apparent from lines 8-22 and 48-60. Subsequent transitions between the states can be seen in the last sections of *extended interface automata* (lines 23-38 and 61-78).

The classification of operations is a key element of *extended interface automata*, requiring careful judgement during translation. Given the sepafration, however, we propose an improved Contract Definition View in the Contract Pattern. As shown in Figure 4, the Contract Definition Viewpoint identifies the input, output and hidden operations, which makes this explicit in the SysML model, and provides independence from the Interface Pattern.

In contrast to CML, during the contract translation to the interface automaton, the software engineer is led to the division of methods based on their type (input, output, hidden operations), thereby increasing the transparency. With a system described in this way we can perform classified verification operations, such as determination that constituent systems are composable (or not) by checking conditions on the actions' viability by considering their semantics. Furthermore, considering the synchronized product, we are able to determine inconsistencies between the sequences of action calls given by communicating protocols.

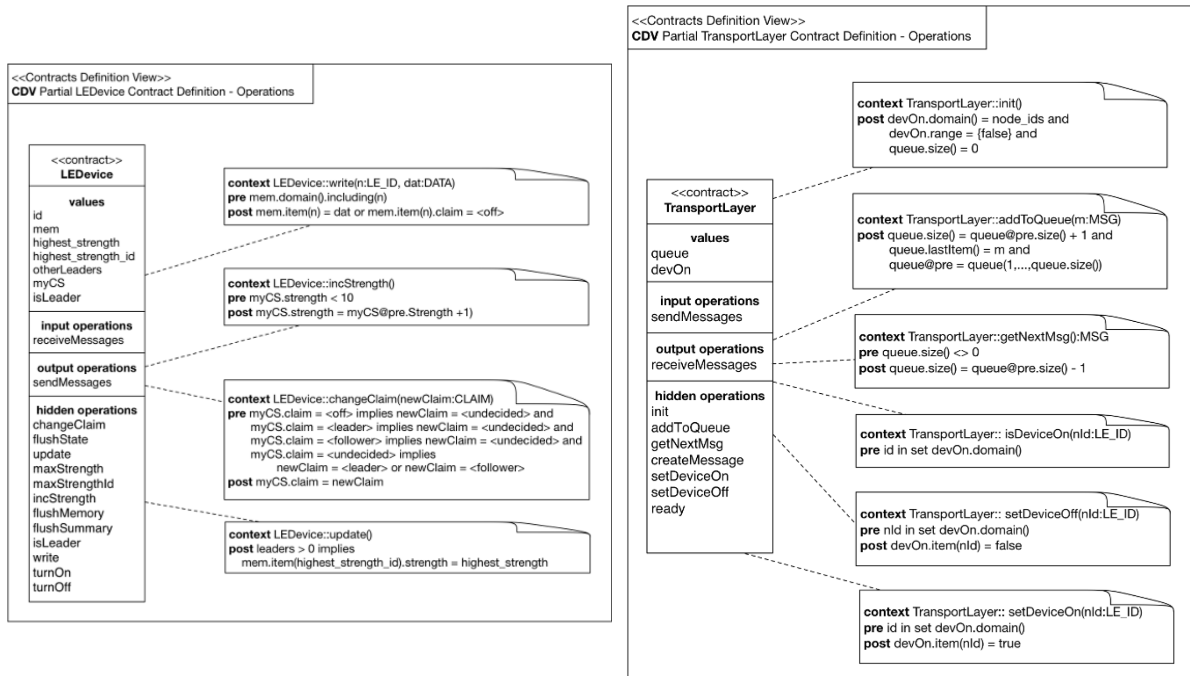

Figure 4: Improved Contract Definition View by separation operations

7 Conclusion

This paper proposes an extension of the Contract Pattern to analyse the compatibility between contracts. The paper contribution forms two main parts. Firstly, we proposed a substitution of CML for OCL language (Object Constraint Language) which is used in order to extend the SysML in the Contract Pattern. The use of OCL provides a more natural fit with the base SysML views. In addition, we improved the Contract Pattern by separation of the input, output and hidden operations, which increased its transparency and independence on the Interface Pattern.

Secondly, based upon the contracts defined in the Contract Pattern, we propose an improvement in interface automata. Originally designed by L. Alfaro and T. Henzinger [1] and modified by Samir Chouali et al. [8], we adapted and extended the interface automata approach by values of the contract to represent its states. Linking CS contracts to interface automata enables the verification of contract composition.

Future Work Given the contributions of this paper, we consider several areas of future work. Firstly, the verification of *LE Device* and *Transpost Layer extended interface automata* compatibility must be demonstrated, based upon the set of definitions in this paper. Given manual verification, we consider automated verification. This implemention may be included as an external plug-in to the Symphony tool, and thus extended the possibilities of performing contract composition analysis in this tool platform. In addition, as mentioned in Section 4.1, the OCL expressions in this paper are defined as SysML notes. Future work would consider the conformance of CSs according to these contract specifications. Finally, we would also like to consider the application of the Contract Pattern and *extended interface automata* in the specification and analysis of evolution and dynamic reconfiguration of SoSs.

Acknowledgments

This work is supported by the INTO-CPS project funded by the European Commissions Horizon 2020 (grant agreement 644047, http://into-cps.au.dk) and by the CPSE Labs funded by the European Union under the Smart Anything Everywhere initiative (http://www.cpse-labs.eu).

References

[1] Luca de Alfaro & Thomas A. Henzinger (2001): *Interface Automata*. SIGSOFT Softw. Eng. Notes 26(5), pp. 109–120, doi:10.1145/503271.503226.

[2] N. Amalio, R. Payne, A. Cavalcanti & E. Brosse (2015): *Foundations of the SysML profile for CPS modelling*. Technical Report, INTO-CPS Deliverable, D2.1a.

[3] A. Arnold, B. Boyer & A. Legay (2013): *Contracts and Behavioral Patterns for SoS: The EU IP DANSE approach*. In Kim G. Larsen, Axel Legay & Ulrik Nyman, editors: Proceedings 1st Workshop on Advances in Systems of Systems, Rome, Italy, 16th March 2013, Electronic Proceedings in Theoretical Computer Science 133, Open Publishing Association, pp. 47–66, doi:10.4204/EPTCS.133.6.

[4] J. Bryans, J. Fitzgerald, R. Payne, A. Miyazawa & K. Kristensen (2014): *SysML contracts for systems of systems*. In: System of Systems Engineering (SOSE), 2014 9th International Conference on, doi:10.1109/SYSOSE.2014.6892466.

[5] J. Bryans, R. Payne, J. Holt & S. Perry (2013): *Semi-formal and formal interface specification for system of systems architecture*. In: Systems Conference (SysCon), 2013 IEEE International, pp. 612–619, doi:10.1109/SysCon.2013.6549946.

[6] Jeremy Bryans, John Fitzgerald, Richard Payne & Klaus Kristensen (2014): *2.2.2 Maintaining Emergence in Systems of Systems Integration: a Contractual Approach using SysML*. INCOSE International Symposium 24(1), pp. 166–181, doi:10.1002/j.2334-5837.2014.tb03142.x.

[7] Michel De Champlain (1997): *The Contract Pattern*. In: In Proceedings of Pattern Languages of Program Design 4 (PLoPD4), doi:10.1.1.38.3112.

[8] S. Chouali, H. Mountassir & S. Mouelhi (2010): *An I/O Automata-based Approach to Verify Component Compatibility: Application to the CyCab Car*. Electronic Notes in Theoretical Computer Science 238(6), pp. 3 – 13, doi:10.1016/j.entcs.2010.06.002.

[9] J. S. Dahmann & K. J. Baldwin (2008): *Understanding the Current State of US Defense Systems of Systems and the Implications for Systems Engineering*. In: 2008 2nd Annual IEEE Systems Conference, pp. 1–7, doi:10.1109/SYSTEMS.2008.4518994.

[10] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat & M. Verhoef (2005): *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, doi:10.1007/b138800.

[11] C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[12] C. Ingram, R. Payne, J. Fitzgerald & L.D.Couto (2015): *Model-based Engineering of Emergence in a Collaborative SoS: Exploiting SysML and Formalism*. INCOSE International Symposium 25(1), pp. 404–419, doi:10.1002/j.2334-5837.2015.00071.x.

[13] Claire Ingram, Richard Payne & John Fitzgerald (2015): *Architectural Modelling Patterns for Systems of Systems*. INCOSE International Symposium 25(1), pp. 1177–1192, doi:10.1002/j.2334-5837.2015.00123.x.

[14] ISO/IEC/IEEE (2015): *15288:2015 Systems and software engineering – System life cycle processes*.

[15] M.W. Maier (1998): *Architecting Principles for Systems-of-Systems*. Systems Engineering 1(4), pp. 267–284, 10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D.

[16] B. Meyer (1988): *Object-oriented Software Construction*. Prentice-Hall International.

[17] Bertrand Meyer (1992): *Applying "Design by Contract"*. Computer 25(10), pp. 40–51, doi:10.1109/2.161279.

[18] Bertrand Meyer (1993): *Systematic Concurrent Object-oriented Programming*. Commun. ACM 36(9), pp. 56–80, doi:10.1145/162685.162705.

[19] C.B. Nielsen, P.G. Larsen, J. Fitzgerald, J. Woodcock & J. Peleska (2015): *Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions*. ACM Comput. Surv. 48(2), pp. 18:1–18:41, doi:10.1145/2794381.

[20] OMG (2012): *OMG Object Constraint Language (OCL), Version 2.3.1*. Available at `http://www.omg.org/spec/OCL/2.3.1/`.

[21] OMG (2015): *OMG Systems Modeling Language (OMG SysML™)*. Technical Report Version 1.4, Object Management Group. Http://www.omg.org/spec/SysML/1.4/.

[22] R. Payne & J.S. Fitzgerald (2010): *Evaluation of architectural frameworks supporting contract-based specification*. Technical Report, Newcastle University.

[23] S. Perry (2013): *Report on Modelling Patterns for SoS Architectures COMPASS Deliverable, D22.3, Tech. Rep.* Available at `http://www.compass-research.eu/deliverables.html`.

[24] S. Perry, J. Holt, R. Payne, J. Bryans, C. Ingram, A. Miyazawa, L.D. Couto, S. Hallerstede, A.K. Malmos, J. Iyoda, M. Cornelio & J. Peleska (2014): *Final Report on SoS Architectural Models*. Technical Report, COMPASS Deliverable, D22.6. Available at http://www.compass-research.eu/.

[25] J. Warmer & A. Kleppe (2003): *The Object Constraint Language: Getting Your Models Ready for MDA*, 2 edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[26] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa & S. Perry (2012): *Features of CML: A formal modelling language for Systems of Systems*. In: 2012 7th International Conference on System of Systems Engineering (SoSE), pp. 1–6, doi:10.1109/SYSoSE.2012.6384144.

[27] J. Woodcock, P.G. Larsen, J. Bicarregui & J. Fitzgerald (2009): *Formal Methods: Practice and Experience*. ACM Computing Surveys 41(4), pp. 1–36, doi:10.1145/1592434.1592436.

[28] Jim Woodcock & Ana Cavalcanti (2002): *The Semantics of Circus*, pp. 184–203. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/3-540-45648-1_10.