Explicit Connection Patterns (ECP) Profile and Semantics for Modelling and Generating Explicit Connections in Complex UML Composite Structures

Federico Ciccozzi

School of Innovation, Design and Engineering (IDT)
MRTC, Mälardalen University
Västerås, Sweden

Abstract

Model-driven engineering can help in mitigating ever-growing complexity of modern software systems. In this sense, the Unified Modeling Language (UML) has gained a thick share in the market of modelling languages adopted in industry. Nevertheless, the generality of UML can make it hard to build complete code generators, simulators, model-based analysis or testing tools without setting variability in the semantics of the language. To tailor semantics variability the notion of semantic variation point has been introduced in UML 2.0.

Our research focuses on the semantic variation point that leaves the rules for matching multiplicities of connected instances of components and ports undecided in UML composite structures. In order to allow model analysability, simulation and code generation, this semantics needs to be set. At the same time, leaving the burden of this task to the developers is often overwhelming for complex systems.

In this article we provide a solution for supporting modelling and automatic calculation and generation of explicit interconnections in complex UML composite structures. This is achieved by (i) defining a set of connection patterns, in terms of a UML profile, and related semantic rules for driving the calculation, (ii) providing a generation algorithm to calculate the explicit interconnections.

Keywords: model-driven engineering, UML, composite structure, explicit interconnections, semantic variation point, intentional interpretation, UML profile

1. Introduction

Growing complexity of modern software systems is more and more often tackled by shifting the focus of the development from coding to modelling through the adoption of, e.g., Model-Driven Engineering (MDE). While modelling and abstraction can help in mitigating complexity [1], MDE can achieve a dominant position in industrial
development only if modelling tasks are proven to be more efficient than coding activities. In fact, if the complexity of model-driven development tasks was comparable with the ones of code-centric development, the return of investment could be unsatisfactory to justify such a technology shift [2]. This is especially relevant in industrial processes, where efficiency of the development is of paramount importance.

In the gradual adoption of MDE in industry that the community has been experiencing in the last 15 years, the Unified Modeling Language (UML) has played a central role together with its plenitude of different tools, both commercial and free-ware [3]. At the same time the generality of UML, while enabling the modelling of systems in theoretically any application domain, makes it hard to build complete code generators, model simulators, model-based analysis or testing tools without fixing variability points in the semantics of the language. To permit the developer to tailor semantics variability for her specific purposes the notion of semantic variation point has been introduced in UML 2.0 [4]. A semantic variation point is defined as a point of variation in the semantics of the UML metamodel and provides an intentional degree of freedom for its interpretation. Thanks to semantic variation points, UML was meant to become a family of languages with commonalities as well as variabilities that could be tailored to a given application domain or problem [5].

Concerning modelling of software systems, UML provides several different alternatives. One of them follows the component-based fashion where a system is defined as an assembly of components communicating via interfaces (exposed by ports) [6]. Moreover, multiple instances of components can provide features via ports to multiple instances of both peer (at the same hierarchical level) and internal components (parts internal to the component providing the features). While the number of instances of components and ports can be precisely specified through multiplicities1 connectors are not equipped with a detailed specification of the component and ports instances they connect. The UML metamodel leaves the rules for matching the multiplicities of connected instances of components and ports as a semantic variation point [6]. However, for analysis purposes, deployment issues, and code generation, it is often preferable to precisely know how those connectors would eventually be realised at instance level. In order to be able to calculate the explicit set of interconnections between instances, precise semantics is needed.

In this article we focus on this specific semantic variation point and provide:

1. **Connection patterns and semantic rules**: in order to be able to give a semantics to the interconnections among component and port instances, boundaries have to be set for their syntactical definition. More specifically, we identified a set of five meaningful combinations of multiplicities (addressed as connection patterns). For each of them we defined specific semantic rules to be deterministically applied for calculating explicit interconnections;

2. **ECP profile**: in order to let the developer decide upon which connection pattern to apply, we propose the Explicit Connection Patterns (ECP) profile for UML that, for each of the connection patterns, provides a specific stereotype. More-

---

1 In this article we employ the terms *multiplicity* and *cardinality* as synonyms.
over, it includes validity rules defined as OCL expressions for validating the application of stereotypes on UML models;

3. **Generation algorithm:** to exploit connection patterns and related semantic rules, we provide a generic algorithm that, from a component-based UML model as input, produces a new version of it obtained by unrolling parts and ports according to their multiplicity as well as by generating explicit interconnections among the unrolled elements. The algorithm is generic in the sense that it is independent of both the output format as well as the technology used for implementing it. A proof-of-concept of the algorithm in terms of an M2M transformation mechanism is provided too.

A preliminary study that inspired this research was presented in [7]. Nevertheless, the delta brought by this contribution is significant and consists of the following.

- The possibility for the user to drive the generation of explicit connections at modelling level thanks to a specific UML profile that provides stereotypes and validity constraints for each of the identified connection patterns;
- The five connection patterns defined in [7] are condensed into two generic patterns, namely *array* and *M-parts–N-ports*. Moreover, the *M-parts–N-ports* connection pattern provides even a solution for the two most complex multiplicity cases that were left out in [7];
- *Star*, *perfect shuffle* [8], and a redefined *redundancy* connection patterns have been introduced in order to allow condensed modelling of complex topologies (e.g., multi-stage logarithmic network topologies for multiprocessor computing such as (multi-)Benes, (multi-)Butterfly and Omega (even reversed)) [9];
- Semantic rules identify both target part and port, while previously focusing simply on target parts, since the output format did not entail information on target ports. This led to an entire re-definition of the generation algorithm;
- While in [7] we targeted non-UML artefacts for storing explicit elements, this contribution targets UML, providing as possible output of the generation algorithm a new UML model that keeps intact hierarchical containments while unrolling parts, ports and connectors according to their multiplicity;
- In order to provide a proof-of-concept, we defined a new M2M transformation implementing the generation algorithm.

Moreover, in this work we relax the constraints by abstracting from the port type (i.e., provided or required) in order to make the solution applicable to plain UML models (rather than only UML+MARTE). The generation algorithm itself has been entirely re-defined in order to entail the newly introduced connection patterns and the re-defined ones, as well as the relaxed constraints regarding port types.

The remainder of this article is organised as follows. In Section 2 we described the main concepts upon which the contribution is based. The problem is formalised in terms of motivation and assumptions in Section 3, while a summary of the related efforts documented in the literature is provided in Section 4. The definition of connection patterns and semantic rules for the calculation of explicit interconnections is provided in Section 5, together with the ECP profile for UML. The generation algorithm and its application to a sample UML model as proof-of-concept are described in Section 6. Details on the evaluation of the approach as well as a discussion on the contribution
are described in Section 7 together with possible future enhancements of the proposed solution. The article is concluded with final remarks in Section 8.

2. Background

The core concept in MDE is the model, considered as an abstraction of the system under development. Rules and constraints for building models have to be properly described through a corresponding language definition and, in this respect, a metamodel describes the set of available concepts and well-formedness rules a correct model must conform to [10].

Following the MDE paradigm, a system is developed by designing models and refining them starting from higher and moving to lower levels of abstraction until code is generated; refinements are performed through model manipulations (so called transformations). A model transformation translates a source model to a target model while preserving their well-formedness [11]. More specifically, in this research work we exploit model-to-model (M2M) transformations for providing automation mechanisms.

When developing with MDE techniques, systems can be modelled in many different ways. In this work we exploit component-based design as prescribed by the UML Superstructure [6]. That is to say, a system is modelled as an assembly of components communicating via interfaces exposed by ports, where a port represents an interaction point between a classifier instance and its internal or external environment. Additionally, features owned by interfaces are meant to be offered by one or more instances of the owning classifier to one or more instances of the classifiers in its internal or external environment.

In the following, these concepts are described in a simplified manner entailing only the details needed for grasping the contribution of this work. More details can be found in the formalisation of the concepts provided in Appendix A.

A component represents a modular constituent of the system which encapsulates its internal environment [6]. We formalise a component as a tuple $C = \langle pr\{\}, p\{\}, con\{\} \rangle$, where $pr\{\}$ represents a set of owned parts, $p\{\}$ represents a set of owned ports, and $con\{\}$ represents a set of owned connectors.

A part is a role played by a set of instances of a component at runtime; a part can include a multiplicity factor [6]. We formalise a part as a pair $pr = \langle C, M_{pr} \rangle$, where $C$ is a component and represents the part’s type, and $M_{pr}$ represents the part’s concise multiplicity, with $M_{pr} \geq 1$. An explicit part is a role played at runtime by exactly one instance of a component. We introduce this element in order to be able to unambiguously represent explicit interconnections among parts representing sets of instances when $M_{pr} > 1$. An explicit part is formalised as $pr_{ipr} = \langle C, 1 \rangle$ where $i_{pr} \in [1, M_{pr}]$.

A port is an interaction point which is used to connect structured components with their parts as well as with the surrounding environment and it may contain a multiplicity factor representing the number of port instances at runtime [6]. We formalise a port as a pair $p = \langle I, M_p \rangle$, where $I$ represents the ports type (i.e., an interface) and $M_p$ represents the port’s concise multiplicity, with $M_p \geq 1$. Moreover, multiplicity represents $M_p$. An explicit port represents a single port instance at runtime and it is
introduced for representing explicit interconnections when a port \( p \) has \( M_p > 1 \). This element is formalised as \( p_{i_p} = \langle I, I \rangle \) where \( i_p \in [1, M_p] \).

Connections between ports are achieved through connectors, which can be of two kinds: delegation and assembly. A delegation connector represents the forwarding of events meaning that what arrives at a port that has a delegation connector to one or more parts or ports on parts will be passed on to those targets for handling. An assembly connector links two or more parts or ports on parts (at the same hierarchical level) that defines that one or more parts provide the services that other parts use. A connector is formalised as a tuple \( \text{conn} = \langle T, ce_1, ce_2 \rangle \), where \( T \in \{\text{assembly, delegation}\} \) represents the connector’s type and the two connector ends are represented by \( ce_1 \) and \( ce_2 \). An explicit connector is defined as a tuple \( \text{conn}_j = \langle T, \langle \text{pr}_1, \text{p}_1 \rangle, \langle \text{pr}_2, \text{p}_2 \rangle \rangle \), where \( j \in [1, M_{pr} \times M_p] \) and represents one element in the set of explicit interconnections represented by \( \text{con} \).

Figure 1 depicts a UML composite structure diagram model presented in a visual concrete syntax where the above introduced concepts are used. More specifically, we can see the main component \( A = \langle \{f, d\}, \{\text{ap}\}, \{\text{conn}_f\_d, \text{conn}_a\_f\} \rangle \) composed of:

- Parts: \( f = \langle F, 2 \rangle \), \( F \) owns ports \( fp = \langle \text{PortType}, 3 \rangle \) and \( ffp = \langle \text{PortType}, 2 \rangle \); \( d = \langle D, 3 \rangle \), \( D \) owns port \( dp = \langle \text{PortType}, 2 \rangle \);
- Ports: \( \text{ap} = \langle \text{PortType}, 4 \rangle \);
- Connectors: \( \text{conn}_f\_d = \langle \text{assembly}, \langle f, fp \rangle, \langle d, dp \rangle \rangle \) and \( \text{conn}_f\_a = \langle \text{delegation}, \langle f, ffp \rangle, \langle \text{null}, \text{ap} \rangle \rangle \).

Figure 1: Assembly and delegation in UML and Explicit

3. Problem formalisation

In this section the problem is formalised in terms of motivation and the assumptions needed in order to provide a feasible and technically sound solution.

3.1. Motivation

When defining the system in terms of UML components and ports, minimum and maximum allowed or expected amount of instances (i.e., multiplicity) can be specified
for each of these elements; as explained later in this section, we only consider multiplicities as concise values (i.e., \([n]\)). Figure 1 depicts a motivational example of assembly and delegation connections.

On the one hand, parts \(f\) and \(d\), as well as component \(A\) and part \(f\), are linked through connectors that do not entail any information on the explicit instance-to-instance interconnections. This means that designers actually have a precise idea of the topologies they model only when the two connected ports have the same multiplicity as well as the related parts \([8]\). All other cases cannot be considered as deterministic nor self-explanatory. On the other hand, the determinism of this information is a prerequisite for several activities, from the accurate analysis of important system properties at modelling level (e.g., analysis of UML composite structure diagram) to model simulation as well as for the generation of the full-fledged target code.

Moreover, while the intentional freedom given by UML allows to model complex topologies in a very condensed way, it is far from straightforward for the developer to capitalise the most on it without knowing how decisions taken at modelling level affect the actual realisation at runtime.

Information on the explicit instance-to-instance interconnections might be modelled by-hand in different ways at structural level through, e.g., OCL constraints or ad-hoc annotations, or even at behavioural level through action languages. In any case, when dealing with complex systems, the manual effort demanded by this task becomes soon overwhelming and error-prone, two weaknesses not tolerable in industry. For instance, let us consider the Asynchronous Transfer Mode Adaptation Layer 2 subsystem exploited in Section 6.2. With its hundred thousands of implicit component and port instances and multiple levels of hierarchical composition of components, manual modelling of explicit instances and interconnections between would be extremely tedious and far from flawless.

Another option could be to manually define this information at code level; different drawbacks, such as unintentional injection of errors, inconsistency between models and code, as well as the often excessive effort needed for carrying out the task make this solution inappropriate. For all these reasons, the UML semantic variation point related to this issue should be settled through the specification of a precise semantics through which the developer can benefit from a deterministic and automatic calculation of the explicit interconnections among instances. On the right-hand side of Figure 1, the general idea of driving and generating explicit interconnections is applied to the connectors depicted on the left-hand side. More specifically parts and ports are unrolled according to their multiplicity and explicit interconnections among them are created following the semantic rules we propose in Section 5.

3.2 Assumptions

In our solution we only entail those cases in which multiplicities are defined as concise values (i.e., \([n]\)) while range values (e.g., \([m..n]\)) are left as a possible future

---

\[ \text{Note that we graphically stay at the level of parts without involving UML instance specifications in order to be able to show explicit connectors for the sake of clarity. Nevertheless this does not prevent the replacement of unrolled elements with instance specifications, in the UML sense.} \]
enhancement. Moreover, a further assumption has been made in regards to intercon-
nections among components: connectors are untyped, meaning that the multiplicity of
their connector ends is equal to the one of the related port (i.e., role). While in [7] we
only allow one binary interconnection per port, in this solution we provide a solution
for multiple binary interconnections, meant as multiple connectors sharing ports.

Anyhow, these assumptions did not hinder us from being able to model complex
industrial systems, but were rather set to circumscribe the scope of the problem and
propose a technically sound solution. Clearly, since no unique interpretation of the
UML metamodel semantics can be given for semantic variation points, we focused on
our scope with particular attention to the issues related to the generation of full-fledged
code, those related to deployment in terms of allocation of component instances to tasks
and tasks to hardware resources as well as the needs of specific model-based analysis
techniques.

4. Related work

In this section we compare our contribution to similar research works documented
in the literature. Overall, our contribution is meant to advance the state-of-the-art re-
garding the explicitation and the automatic generation of interconnections between
component and port instances in UML composite structures. More specifically, cur-
tently these interconnections are either modelled manually based on the modeller’s
experience or left aside to be programmed at code level. With our contribution, we
provide a novel UML profile with constraints for fixing the semantic variation point
related to component interconnections through a set of semantic rules. These rules are
deterministically leveraged by a novel algorithm that automatically generates explicit
interconnections with no need of manual user intervention.

4.1. Semantic Variation Points and Ambiguity of Connections in UML

The concept of component together with a specific diagram for it have been intro-
duced in the UML 2.0 [6]. Bock [12] addresses composition mechanisms provided
with UML 2.0 focusing on the role of multiplicities in interconnections among com-
posite structures. Nonetheless, the author focuses on the issue of realising component
interconnections at different abstraction levels hence leaving aside the problems related
to concrete instantiation, which is what we focus on. Similar works address different
semantic aspects in composite structures, such as [13] where the authors tackle the
problem of request propagations across ports.

Generally, several works in the literature aim attention at the issues related to se-
monic variation points, such as [5] where the authors exploit those concerning syn-
chronicity in state-machines towards code generation using the KerMeta metalanguage
[14] for fixing semantics, or [15] where the authors provide matters to disambiguate
variation points related to other aspects of UML diagrams than the one we address.

Cuccuru et al. [8] provide a solution to the MARTE RFP requirement concerning
the definition of common high-level modeling mechanisms for factorizing repetitive
structures. We agree with author’s argument that designers can only have a precise
idea of the component-based topologies they model in UML when the multiplicities of
connector ends are equal to 1 and the ones of the two roles match. This consideration motivated the research work presented in this article where we provide (i) deterministic ways to tackle situations in which the multiplicities of the two roles do not match (ii) as well as a set of different alternatives for the two topologies that the authors address.

In [16], the author demonstrates how semantic intentional interpretation of associations improves expressiveness of the modelling language and shows other interesting advantages. The paper focuses on the concepts of association ends, associations and their symmetry, while our goal is to address the issues related to component-based design (e.g., hierarchical composition, encapsulation). Moreover, we propose both the possibility of multiple interpretations of the same connections in composite structures as well as an automatic mechanism for generating them.

Other research works are dedicated to settle semantics across links through ports and keeping its correctness [17] [18] [19]; anyhow they focus on interface definitions (in terms of type and behaviour of ports and connectors) rather than on the issue of explicitly calculating instance-to-instance interconnections, which represents our goal. Even component-based design tool implementations, as the solution proposed in [20], neglect the problem we are addressing in this work.

4.2. Automatic Generation in Modelling Tools

When it comes to the tools providing code generation from UML models, different solutions are provided for the generation of interconnections between component instances. Enterprise Architect [21], by Sparx Systems, provides code generation from class diagrams where classes are linked through associations. In this case code is generated in a way that instances of the association’s target class are owned by the instances of the source class. In our solution we target component-based design for encapsulation reasons and we aim at generating code which preserves this paradigm, that is to say components communicating by invoking functionalities on their own ports with no need of knowing which component is providing the functionality on the other end of the connector. In this way generated code can be consistent to what specified at modelling level in terms of components and preservation of system properties from models to code is facilitated [22].

IBM Rational Rhapsody [23] keeps the generality of UML when coming to matching the multiplicities of components and ports. No decision is in fact automated about the interconnections between components via ports, but rather instances are generated according to their multiplicities along with function handlers (i.e., get and set) for managing connections when needed. In this case, the developer needs to specify how to connect the different component instances when describing the behaviour of the single components.

Our solution is based on the interpretation of the UML metamodel’s semantics, as prescribed by the notion of semantic variation point, in order to automate the generation of explicit interconnections between component and port instances whose manual specification would require heavy and error-prone modelling effort in case of complex systems. In fact, while the absence of automation in the generation of explicit interconnections does not affect the validity of analysis, simulation, and code generation techniques, it can negatively affect scalability as well as error-proneness.
5. Connection patterns and semantics for explicit interconnections

The guidelines given by the superstructure specification of UML do not provide a fixed semantics concerning the multiplicities of interconnected components and ports. This means that any combination of multiplicities is syntactically allowed and therefore syntactically correct. In order to be able to give a semantics to the interconnections among instances, boundaries have to be set for their syntactical definition. In other words, only a set of meaningful combinations of multiplicities of connected parts and ports are considered in order for the semantic rules to be deterministically applied.

In the case of an assembly connector \( \text{conn} = \langle \text{assembly}, \langle a, \text{ap} \rangle, \langle b, \text{bp} \rangle \rangle \) between two parts \( a \) and \( b \), their multiplicity is expressed respectively as \( M_a \) and \( M_b \). An instance of part \( a \) is represented by \( a_{i_a}, i_a \in [1, M_a] \). Concerning part \( b \), an instance is represented by \( b_{i_b}, i_b \in [1, M_b] \). Moreover, part \( a \) is connected through its port \( \text{ap} \) to port \( \text{bp} \) of part \( b \); the multiplicities of these ports are expressed respectively as \( M_{\text{ap}} \) and \( M_{\text{bp}} \). An instance of port \( \text{ap} \) is represented by \( \text{ap}_{i_{\text{ap}}}, i_{\text{ap}} \in [1, M_{\text{ap}}] \). Regarding port \( \text{bp} \), an instance is represented by \( \text{bp}_{i_{\text{bp}}}, i_{\text{bp}} \in [1, M_{\text{bp}}] \).

From a connector \( \text{conn} = \langle \text{assembly}, \langle a, \text{ap} \rangle, \langle b, \text{bp} \rangle \rangle \), the aim is to automatically calculate (and generate) the set of explicit connectors in the form \( \text{conn}_j = \langle \text{assembly}, \langle a_{i_a}, \text{ap}_{i_{\text{ap}}} \rangle, \langle b_{i_b}, \text{bp}_{i_{\text{bp}}} \rangle \rangle \) with \( j \in [1, M_a \times M_{\text{ap}}] \). In order to create explicit connectors, the basic idea is to iterate on pairs \( \langle a_{i_a}, \text{ap}_{i_{\text{ap}}} \rangle \) in order to calculate, through the indices \( i_a \) and \( i_{\text{ap}} \), the indices \( i_b \) and \( i_{\text{bp}} \) to identify the other connector end \( \langle b_{i_b}, \text{bp}_{i_{\text{bp}}} \rangle \).

Note that the delegation case is considered, in terms of combination of multiplicities, as a particular case of the assembly, where one of the parts (e.g., \( b \)) is a component \( C \) rather than a part, and therefore we consider its multiplicity as \( M_C = 1 \). In this particular case, the explicit connectors will have the following form: \( \text{conn}_j = \langle \text{delegation}, \langle a_{i_a}, \text{ap}_{i_{\text{ap}}} \rangle, \langle \text{null}, \text{bp}_{i_{\text{bp}}} \rangle \rangle \).

5.1. Connection patterns and related semantics

In order for the generation process to be able to calculate the explicit connections, a general condition that shall hold for each connection is: each port instance of each port instance or component should be connected to one port instance owned by a port instance or component. This rule is represented in terms of multiplicities for a single connector by the condition in Equation 1 that shall therefore always be true for each connector \( \text{conn} = \langle T, \langle a, \text{ap} \rangle, \langle b, \text{bp} \rangle \rangle \). Our approach covers any combination of multiplicities which satisfies the condition; all the other possible combinations are not entailed since semantic rules cannot be deterministically applied to them. The simplified concrete graphical syntax depicted in Figure 2 is used to explain the connection patterns.

\[
M_a \times M_{\text{ap}} = M_b \times M_{\text{bp}} \tag{1}
\]

We defined a set of five connection patterns, and related semantic rules, for creating assembly and delegation explicit connectors based on a set of multiplicity combinations described in the following paragraphs. In the modelling environment (see Section 6) we
more details) connector ends are ordered according to their creation order. For minimising the number of rules, we provide the order needed for each specific connection case for the rules to be applied symmetrically, disregarding the original order. Moreover, note that, during the generation process, the semantic rules can be applied in any order producing the same result.

Figure 2: Simplified concrete graphical syntax

**Case 1: Array connector pattern.** The array connector pattern can be applied when \( a \) and \( b \) have the same multiplicity as well as the connecting \( ap \) and \( bp \) ports. When calculating explicit connections, the two connector ends \((a_i, ap_i)\) and \((b_i, bp_i)\) are “mirrored” in terms of their respective indices as follows:

\[
(M_a == M_b) \land (M_ap == M_bp) \implies \begin{cases} 
    i_b = i_a \\
    i_bp = i_ap 
\end{cases}
\]

(Array connector pattern)

On the left-hand side of Figure 3 we show a sample composite structure before and after the application of the array connection pattern.

**Case 2: Redundancy pattern.** In the specific case where \((M_a == M_bp = n) \land (M_b == M_ap = m)\) with \((n > 1) \land (m > 2)\), the user can choose to apply the redundancy pattern instead. In this way, explicit connectors are generated in a way that each port instance \(ap_{i_{ap}}\), of the part \(a_i\), is connected to a different part \(b_i\) through \(bp_{i_{bp}}\). This pattern is useful to implement redundancy, for the purpose of increasing reliability of
the system, and model it in a condensed manner as shown on the right-hand side of Figure 3 where a sample composite structure before and after the application of the redundancy pattern are depicted. The rules for identifying \( (i_b, i_{bp}) \) are:

\[
(M_a == M_{bp} = n) \land (M_b == M_{ap} = m) \land (n > 1) \land (m > 2) \Rightarrow \begin{cases} i_b = i_{ap} \\ i_{bp} = i_a \end{cases}
\]

(Redundancy pattern)

**Case 3: Star connector pattern.** A special case of the redundancy pattern is when \( M_a == M_b = 2n \) with \( n \geq 1 \) and \( M_{ap} == M_{bp} == 2 \). In this situation, the user can choose to apply the star connector pattern which can be used to design well-known multi-stage logarithmic network topologies for multiprocessor computing such as (multi-)Butterfly, (multi-)Benes (depicted in Figure 4) and Banyan networks [9].

When \( M_a == M_b = 2^n \) with \( n \geq 2 \), the pattern can even specify alternate star topologies, so called butterflies [9]. To do so, the user selects the wished size of the alternation such that \( size \geq 1 \) with \( (size \times 2) \leq M_a \), as depicted in Figure 4. Note that in case \( (n \leq 2) \lor ((M_a == M_b == 2^n) \land ((size \times 2) > M_a)) \), alternation (butterflies) cannot be generated, thereby we consider \( size = 1 \).

![Figure 4: Star connector pattern (e.g., for Butterfly and Benes networks)](image)

The star connector pattern aims at identifying an alternate crossing pairwise connection (i.e., 4 explicit connectors) between the ports of two instances of \( a \) and the ones of two instances of \( b \) at a time as depicted in Figure 4. In order to do so, at each iteration, starting from the ordered set of indices \( i_a \), with \( i_a \in [1, M_a] \), we extract the immediately following sibling of the previous extracted \( i_a \), addressed as \( i_{a1} \). At this point, the indices of the second instance of \( a \) is identified as \( i_{a1} + size \) and those of the two
instances of $b$ and their ports can be derived as:

$$(M_a == M_b = 2n) \land (n \geq 1) \land (M_{ap} == M_{bp} == 2) \land (size \geq 1) \Rightarrow$$

$$\begin{cases} 
  i_b = \{ 
  i_a , i_{ap} == 1 \\
  i_a + size , otherwise 
\} \\
  i_{bp} = \{ 
  1 , i_a == i_{a1} \\
  2 , otherwise 
\} 
\end{cases}$$

(Star connector pattern)

After each iteration $i_{a1}$ and $i_{a1} + size$ are removed from the set of the remaining indices $i_a$ to connect.

**Case 4: Perfect shuffle pattern.** In the specific situation where $(M_a == M_b = 2n) \land (n \geq 2)$, the user can alternatively choose to apply the perfect shuffle pattern [8], depicted in Figure 5. This can be used, e.g., to design multi-stage logarithmic network topologies for multiprocessor computing such as Omega and reversed Omega, and (multi-)Benes (depicted in Figure 5) [9]. Note that, when $M_{ap} == M_{bp} == 2$, the star connector pattern could be applied too; the selection is done by the user, e.g. through selecting the profile’s specific stereotype (see Section 5.2) to apply.

![Figure 5: Regular and reversed perfect shuffle pattern](image)

The user can even choose to apply the reversed perfect shuffle (Figure 5). In this case, instead of identifying indices $i_b$ and $i_{bp}$ from $i_a$ and $i_{ap}$, the process is inverted. That
is to say, starting from \( i_b \) and \( i_{bp} \), the aim is to identify indices \( i_a \) and \( i_{ap} \).

\[
(M_a == M_b == 2n) \land (M_{ap} == M_{bp} == 2) \land (n \geq 2) \Rightarrow \\
\left\{ \\
\begin{array}{l}
  i_b = \left\lceil \frac{(i_a - 1) \times M_{ap} + i_{ap}}{M_b} \right\rceil \\
  i_{bp} = \left\lfloor \frac{(i_a - 1) \times M_{ap} + i_{ap}}{M_{bp}} \right\rfloor \\
\end{array}
\right.
\]

(Perfect shuffle pattern)

**Case 5: M-parts–N-ports pattern.** The M-parts–N-ports pattern covers the cases that satisfy Equation 1 and that are not covered by the previous patterns. More specifically, it applies to the multiplicity combinations that satisfy Equation 2. In this case the connector ends are ordered so that \( M_{ap} \leq M_{bp} \), as shown in Figure 6.

\[
(M_a \neq M_b) \land ((M_a \times M_{ap}) == (M_b \times M_{bp})) \land (M_{a,b,ap,bp} \geq 1)
\]  

(2)

![Figure 6: M-parts–N-ports pattern](image)

When \( (M_a == M_{bp} = n) \land (n > 1) \land (M_b == M_{ap} = m) \land (m > 2) \), the user can choose to apply the redundancy pattern (Case 2).
5.2. ECP: a UML profile for modelling connection patterns

The connection patterns have been synthesized into a specific extension of UML in terms of the ECP profile\(^4\) depicted in Figure 7. The profile has been implemented in the Papyrus modelling environment \([24]\) and it introduces the concept of explicit connector through the abstract stereotype \textit{ExplicitConnector} which extends the UML’s \textit{Connector} metaelement. Each of the connection patterns are defined as a specific stereotype specializing \textit{ExplicitConnector}, as follows:

- \textit{Array} corresponds to the array connector pattern;
- \textit{Star} corresponds to the star connector pattern and, depending on the value of the integer attribute \textit{size} (set to 1 by default), it defines the alternation size of the star (butterfly if \textit{size} > 1);
- \textit{Redundancy} corresponds to the redundancy pattern;
- \textit{PerfectShuffle} corresponds to the perfect shuffle pattern and, in case the boolean attribute \textit{isReversed} == \textit{true}, to the reverse perfect shuffle;
- \textit{MpartsNports} corresponds to the M-parts–N-ports pattern.

Moreover, in order to enable validation of the models to check that stereotypes are correctly applied to suitable connectors in terms of multiplicity combinations, we defined a set of OCL constraints as part of the profile, as shown in Appendix B.

\(^{4}\)The ECP profile implementation can be found at \url{http://www.mrtc.mdh.se/ECPprofile}.

Figure 7: ECP profile
6. Automatic generation of explicit interconnections

The connection patterns and related semantic rules are meant to be employed for generating explicit interconnections among instances. In this section we describe a generic algorithm to produce an unrolled version of an input UML model in terms of components, parts, ports and explicit interconnections. Moreover, we provide a proof-of-concept of the solution implemented as an M2M transformation and applied to a simplified industrial software system model.

6.1. Generation algorithm

Given a connector \( \langle T, \langle a, ap \rangle, \langle b, bp \rangle \rangle \) between two generic parts \( a \) and \( b \) and via their ports \( ap \) and \( bp \), the goal is to automatically calculate the set of explicit connectors in the form \( \langle T, \langle a_{i_{ap}}, ap_{i_{ap}} \rangle, \langle b_{i_{bp}}, bp_{i_{bp}} \rangle \rangle \) with \( i_{ap} \in [1, M_a \times M_{ap}] \). To do so we have defined the generation algorithm shown (in pseudo-code) in Algorithm 1. The algorithm takes a UML model (\( \text{UmlModel} \)) as input and, by exploiting connection patterns and semantic rules defined in the previous section, produces in output a new version of it (\( \text{ExplicitUml} \)) obtained by unrolling parts and ports as well as by generating explicit connectors among them.

The algorithm operates as described in the following paragraphs. Note that the algorithm’s output, depending on the user needs, can be presented in different ways – e.g., plain text, matrices, tables, other UML-based formats or different modelling artefacts.

**Lines 2-4.** This first part of the algorithm recreates the structure of \( \text{UmlModel} \) into \( \text{ExplicitUml} \) in terms of packages and components and creates explicit parts and ports according to their multiplicities. First of all \( \text{UmlModel} \) is navigated and root packages containing components are seamlessly copied to the \( \text{ExplicitUml} \) through the \( \text{instantiatePackage}() \) recursive function.

It creates the package and the contained components. Within a component, both owned parts and ports are created unrolling them according to their multiplicities and added to the newly generated component. More specifically, for a part \( pr = \langle C, M_{pr} \rangle \), a set of explicit parts \( pr_{i_{pr}} = \langle C, 1 \rangle \) are generated where \( i_{pr} \in [1, M_{pr}] \). For a port \( p = \langle I, M_p \rangle \), a set of explicit ports \( p_{i_p} = \langle I, 1 \rangle \) are generated where \( i_p \in [1, M_p] \). The function is recursively called on contained packages in order to keep the original hierarchy intact.

**Lines 5-11.** At this point the algorithm checks connectors in \( \text{UmlModel} \). Each connector \( \langle T, \langle a, ap \rangle, \langle b, bp \rangle \rangle \) is checked to identify its applied stereotype (and thereby connection pattern).

In case no stereotype is applied, a validity check is run on the multiplicities to ensure that their combination obeys to the condition in Equation 1 (line 9). This check is performed in terms of an \( \text{assert} \), meaning that, if the condition is not true, the algorithm fails and immediately stops its execution. Otherwise, \( M_{parts} - N_{ports} \) pattern is applied as default.
Depending on the stereotype, for each \(a_i\) with \(i \in [1, M_a]\) and each \(ap_{i_{ap}}\) with \(i_{ap} \in [1, M_{ap}]\) contained in the newly created component \(c\), we generate a set of explicit connectors applying the related rules. The generated explicit assembly connectors are added to \(c\). For Array and Redundancy, the rules can simply be applied and the connectors created without additional operations.

For Star, a check on the specified size of the alternation must be done (lines 32-36). If the specified alternation cannot be applied then the alternation size is set to 1. At this point, a set \(I_a\) containing all the indices \(i_a \in [1, M_a]\) is created (lines 37-39). Until the set becomes empty, the next \(i_{a,1} \in I_a\) is retrieved (line 41), \(i_{a,2}\) is calculated, and the four connectors building up a star are generated (lines 43-50). The indices \(i_{a,1}, i_{a,2}\) are removed from \(I_a\) (line 51).

For PerfectShuffle, the stereotype’s boolean attribute isReversed is checked and, if true, the rules are applied from the opposite direction (lines 55-64), meant that for each \(b_i\) with \(i \in [1, M_b]\) and each \(bp_{i_{bp}}\) with \(i_{bp} \in [1, M_{bp}]\) contained in the newly created component \(c\), we generate a set of explicit connectors towards \(a_i\) and \(ap_{i}\).

Otherwise the rules are applied in the regular direction (lines 65-76).

For MpartsNports a check has to be performed on the type of connector (line 91) since in this case we may have both assembly and delegation, while in the other cases we always handle assembly connectors. If the type is delegation then \(b_i\) is set to null in the related connector end when creating the explicit connector (line 92).

**Algorithm 1** Algorithm for generation of explicit connectors

1: `Uml2ExplicitUml(in UmlModel, out ExplicitUml){
2: for each package p in UmlModel do
3: newP = p.instantiatePackage();
4: ExplicitUml+ = newP;
5: for each component c in ExplicitUml do
6: for each connector conn = \(\langle T, \langle a, ap \rangle, \langle b, bp \rangle \rangle\) in c do
7: type = conn.getStereotype();
8: if type = null then
9: assert fatal(\(M_a \times M_{ap} === M_b \times M_{bp}\));
10: type.name = MpartsNports;
11: end if
12: switch type.name do
13: case (Array) //Array connector pattern
14: for each index \(i_a\) in \([1, M_a]\) do
15: for each index \(i_{ap}\) in \([1, M_{ap}]\) do
16: \(i_b = i_a\);
17: \(i_{bp} = i_{ap}\);
18: newConn = \(\langle \text{assembly}, \langle a_{i_a}, ap_{i_{ap}} \rangle, \langle b_{i_b}, bp_{i_{bp}} \rangle \rangle\);
19: end for
20: end for
21: end for
22:}`
Algorithm 1 Algorithm for generation of explicit connectors

22: case (Redundancy) //Redundancy pattern
23: for each index $i_a$ in $[1, M_a]$ do
24: for each index $i_{ap}$ in $[1, M_{ap}]$ do
25: $i_b = i_{ap}$;
26: $i_{bp} = i_a$;
27: newConn = $\langle$assembly, $\langle a_{i_a}, a_{i_{ap}} \rangle$, $\langle b_{i_b}, b_{i_{bp}} \rangle$$\rangle$;
28: $c+$ = newConn;
29: end for
30: end for
31: case (Star) //Star connector pattern
32: if ($\log(M_a) \geq 2$) $\land$ ($\text{type.size} \geq 1$) $\land$ ((type.size $\times$ 2) $\leq$ $M_a$) then
33: size = type.size;
34: else
35: size = 1;
36: end if
37: for each index $i_a$ in $[1, M_a]$ do
38: $I_a+$ = $i_a$;
39: end for
40: while $I_a! = \emptyset$ do
41: $i_{a_l}$ = $I_a$.next();
42: $i_{a_2}$ = $i_{a_l}$ + size;
43: newConn1 = $\langle$assembly, $\langle a_{i_{a_l}}, a_{i_{ap}} \rangle$, $\langle b_{i_{a_2}}, b_{i_{bp}} \rangle$$\rangle$;
44: $c+$ = newConn1;
45: newConn2 = $\langle$assembly, $\langle a_{i_{a_l}}, a_{i_{ap}} \rangle$, $\langle b_{i_{a_2}}, b_{i_{bp}} \rangle$$\rangle$;
46: $c+$ = newConn2;
47: newConn3 = $\langle$assembly, $\langle a_{i_{a_l}}, a_{i_{ap}} \rangle$, $\langle b_{i_{a_2}}, b_{i_{bp}} \rangle$$\rangle$;
48: $c+$ = newConn3;
49: newConn4 = $\langle$assembly, $\langle a_{i_{a_l}}, a_{i_{ap}} \rangle$, $\langle b_{i_{a_2}}, b_{i_{bp}} \rangle$$\rangle$;
50: $c+$ = newConn4;
51: $I_a$ = $\{i_{a_l}, i_{a_2}\}$;
52: end while
53: case (PerfectShuffle) //Perfect shuffle pattern
54: if type.isReversed then
55: for each index $i_a$ in $[1, M_a]$ do
56: for each index $i_{ap}$ in $[1, M_{ap}]$ do
57: if $(i_a - 1) \times M_{ap} + i_{ap}$ mod $M_a > 0$ then
58: $i_a = ((i_a - 1) \times M_{ap} + i_{ap}) \mod M_a$;
59: else
60: $i_a = M_a$;
61: end if
62: $i_{ap}$ = $(i_a - 1) \times M_{ap} + i_{ap}$;
63: end for
64: end for
65: else
66: for each index $i_a$ in $[1, M_a]$ do
67: for each index $i_{ap}$ in $[1, M_{ap}]$ do
68: if $(i_a - 1) \times M_{ap} + i_{ap}$ mod $M_a > 0$ then
69: $i_a = ((i_a - 1) \times M_{ap} + i_{ap}) \mod M_a$;
70: else
71: $i_a = M_a$;
72: end if
73: $i_{ap}$ = $(i_a - 1) \times M_{ap} + i_{ap}$;
74: end for
75: end for
76: end if
77: newConn = $\langle$assembly, $\langle a_{i_a}, a_{i_{ap}} \rangle$, $\langle b_{i_b}, b_{i_{bp}} \rangle$$\rangle$;
78: $c+$ = newConn;
Algorithm 1 Algorithm for generation of explicit connectors

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>79:</td>
<td>case (MpartsNports) //M-parts–M-ports pattern</td>
</tr>
<tr>
<td>80:</td>
<td>for each index $i_a$ in $[1, M_a]$ do</td>
</tr>
<tr>
<td>81:</td>
<td>for each index $i_{ap}$ in $[1, M_{ap}]$ do</td>
</tr>
<tr>
<td>82:</td>
<td>$i_b = \lceil (i_a-1) \times M_{ap} + i_{ap} \rceil$;</td>
</tr>
<tr>
<td>83:</td>
<td>if $((i_a-1) \times M_{ap} + i_{ap}) \mod M_{bp} &gt; 0$ then</td>
</tr>
<tr>
<td>84:</td>
<td>$i_{bp} = ((i_a-1) \times M_{ap} + i_{ap}) \mod M_{bp}$;</td>
</tr>
<tr>
<td>85:</td>
<td>else</td>
</tr>
<tr>
<td>86:</td>
<td>$i_{bp} = M_{bp}$;</td>
</tr>
<tr>
<td>87:</td>
<td>end if</td>
</tr>
<tr>
<td>88:</td>
<td>end for</td>
</tr>
<tr>
<td>89:</td>
<td>end for</td>
</tr>
<tr>
<td>90:</td>
<td>if $T == \text{delegation}$ then</td>
</tr>
<tr>
<td>91:</td>
<td>newConn = $\langle \text{delegation}, \langle a_{i_a}, ap_{i_{ap}} \rangle, \langle \text{null}, bp_{i_{bp}} \rangle \rangle$;</td>
</tr>
<tr>
<td>92:</td>
<td>else</td>
</tr>
<tr>
<td>93:</td>
<td>newConn = $\langle \text{assembly}, \langle a_{i_a}, ap_{i_{ap}} \rangle, \langle b_{i_b}, bp_{i_{bp}} \rangle \rangle$;</td>
</tr>
<tr>
<td>94:</td>
<td>end if</td>
</tr>
<tr>
<td>95:</td>
<td>$c++ = \text{newConn}$;</td>
</tr>
<tr>
<td>96:</td>
<td>end for</td>
</tr>
<tr>
<td>97:</td>
<td>end for</td>
</tr>
<tr>
<td>98:</td>
<td>end for</td>
</tr>
<tr>
<td>99:</td>
<td>}</td>
</tr>
</tbody>
</table>

6.2. A Proof of Concept: Applying the Solution to the AAL2 Subsystem

In order to provide a proof-of-concept of the generation algorithm, we implemented an M2M transformation that realises it. Given a UmlModel designed following the component-based design pattern as input, the transformation provides a corresponding explicit version – ExplicitUml in output.

The M2M is implemented using the Operational QVT [25] transformation language and integrated in an Eclipse plugin as an extension of the Papyrus modelling environment. It takes a UmlModel as input, and, following Algorithm 1, it operates as follows. The first step consists in recreating the structure of UmlModel into ExplicitUml in terms of packages and components. For each of the components, the transformation unrolls parts and ports according to their multiplicities and, after that, it can generate explicit connectors among unrolled parts and ports by properly applying connection patterns.

The solution proposed in this work has been validated against industrial case-studies as described in Section 7. More specifically, we leveraged the Asynchronous Transfer Mode (ATM) Adaptation Layer 2 (AAL2) subsystem, originally developed to adapt voice for transmission over ATM and currently used in telecommunications as part of connectivity platform systems. The complete AAL2 subsystem is composed by several hundred thousands of component instances and multiple levels of hierarchical composition of components.

Here we propose a simplified version of the AAL2 which is composed by three main components: (i) NCC, (ii) AAL2RI_Client, (iii) NCIClient. Each of these components has a complex internal structure in terms of composition of other components. In this work we consider only part of the NCC internal structure while AAL2RI_Client and NCIClient are considered as stubbed. NCC is a connection handler providing connectivity services for the establishment/release of communication paths.
between pairs of connection endpoints handled by \texttt{AAL2RI\textunderscore Client}. \texttt{NCIClient} represents an application asking for services provided by \texttt{NCC}. Components communicate through functional interfaces (function calls or message passing depending on the deployment configuration) exposed by their provided ports, which can be of type \texttt{NCC2Client} or \texttt{NCCP}.

A typical connection scenario in the AAL2 subsystem is the establishment of a connection between two end-points placed on the same node. This is a constrained case of a more general network-wide connection where the two end-points reside on different nodes and the communication transits through a number of other intermediate nodes in the network. When \texttt{NCIClient} wants to connect two end-points, a connection setup request is sent to \texttt{NCC}; such request contains information about the end-points. \texttt{NCC} asks for the establishment of a connection segment between the end-points to an external component (not modelled in this case-study). Then it sends a request for each end-point to their respective \texttt{AAL2RI\textunderscore Client} to activate the access to the transport layer. Once both end-points have positively responded, \texttt{NCC} confirms the establishment of the connection to \texttt{NCIClient}.

Input and output of the transformation with focus on the application of the connection patterns are described per component in the following paragraphs.

**AAL2 composite component.** At the top of the hierarchy we find the composite component \texttt{AAL2} = \{\texttt{ncc}, \texttt{nci}, \texttt{aal}\}, \{\texttt{inP}\}, \{\texttt{ncc\_nci}, \texttt{ncc\_aal}, \texttt{in\_ncc}\} that is composed of (depicted in Figure 8):

- **Parts:** \texttt{ncc} = \{\texttt{NCC}, 4\}, \texttt{nci} = \{\texttt{NCIClient}, 4\}, \texttt{aal} = \{\texttt{AAL2RI\textunderscore Client}, 4\};
- **Ports:** \texttt{inP} = \{\texttt{NCC2Client}, 4\};
- **Connectors:** \texttt{ncc\_nci} = \{\texttt{assembly}, \{\texttt{ncc}, \texttt{nccP}\}, \{\texttt{nci}, \texttt{nciP}\}\}, \texttt{ncc\_aal} = \{\texttt{assembly}, \{\texttt{ncc}, \texttt{nccP}\}, \{\texttt{aal}, \texttt{aalP}\}\}, \texttt{in\_ncc} = \{\texttt{delegation}, \{\texttt{null}, \texttt{inP}\}, \{\texttt{ncc}, \texttt{nccPP}\}\}.

Once the transformation is applied, owned parts are unrolled as follows:

- \texttt{ncc} = \{\texttt{NCC}, 4\} as \texttt{ncc\_1}, \texttt{ncc\_2}, \texttt{ncc\_3}, \texttt{ncc\_4};
- \texttt{nci} = \{\texttt{NCIClient}, 4\} as \texttt{nci\_1}, \texttt{nci\_2}, \texttt{nci\_3}, \texttt{nci\_4};
- \texttt{aal} = \{\texttt{AAL2RI\textunderscore Client}, 4\} as \texttt{aal\_1}, \texttt{aal\_2}, \texttt{aal\_3}, \texttt{aal\_4};
The AAL2 component owns port inP which is also unrolled as the set: inP_1, inP_2, inP_3, inP_4. At this point the transformation generates explicit connectors for ncc_ncl, ncc_aal, and in_ncc.

Regarding ncc_ncl = (assembly, ⟨ncc, nccP⟩, ⟨ncl, nclP⟩), the Star stereotype is applied and, depending on the specified size of the alternation, explicit connectors are generated according to the rules specified for the star connector pattern. The resulting connectors ncc_ncl_conn = ⟨assembly, ⟨ncc_ncl, nccP_nclP⟩, ⟨ncl_ncl, nclP_nclP⟩⟩, for size = 1 and size = 2, are shown in terms of indices in Table 1 and depicted in Figure 9.

<table>
<thead>
<tr>
<th>size = 1</th>
<th>size = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_conn</td>
<td>i_ncc</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Indices in ncc_ncl_conn with size = 1 and size = 2

For in_ncc = (delegation, ⟨null, inP⟩, ⟨ncc, nccP⟩), the MpartsNports stereotype (for delegation) is applied and explicit connectors are generated according to the rules specified for the M-parts–N-ports pattern.
The resulting connectors in_ncc_i_conn = ⟨delegation, ⟨ncc_i_ncc, nccPP_i_nccPP⟩, ⟨null, inP_i_inP⟩⟩ are depicted on the left-hand side of Figure 10.

For ncc_aal = ⟨assembly, ⟨ncc, nccP⟩, ⟨aal, aalP⟩⟩, the PerfectShuffle stereotype is applied and explicit connectors are generated according to the rules specified for the perfect shuffle pattern.

Resulting connectors ncc_aal_i_conn = ⟨assembly, ⟨ncc_i_ncc, nccP_i_nccPP⟩, ⟨aal_i_aal, aalP_i_aalP⟩⟩ for the case when isReversed == true are depicted on the right-hand side of Figure 10.

**NCC composite component.** The NCC component has a complex internal structure. In this study we focus on: Dispatcher, which dispatches the incoming connection requests to available node instances NodeElem, that handles management of connections to the network within the single node, and PortHandler, which manages connection resources. Each of these subcomponents has in turn a complex internal structure in terms of components composition; in this simplified version we consider only the first two levels of decomposition (down to the NCC’s internal structure).

Part ncc in the composite component AAL2 is of type NCC, which is in turn a composite component defined as $NCC = \{\text{disp, node, portH}\}, \{nccP, nccPP\}, \{\text{disp, node, disp, port}\}$ (depicted in Figure 11) where:

- Parts: $\text{node} = \{\text{NodeElem, 3}\}, \text{portH} = \{\text{PortHandler, 3}\}, \text{disp} = \{\text{Dispatcher}, 4\}$;
- Ports: $nccP = \{\text{NCC2Client, 2}\}, nccPP = \{\text{NCC2Client, 1}\};$
Once the transformation is applied, owned parts are unrolled as follows:

- \( \text{node} = \langle \text{NodeElem}, 3 \rangle \) as \( \text{node}_1, \text{node}_2, \text{node}_3 \);
- \( \text{portH} = \langle \text{PortHandler}, 3 \rangle \) as \( \text{portH}_1, \text{portH}_2, \text{portH}_3 \);
- \( \text{disp} = \langle \text{Dispatcher}, 4 \rangle \) as \( \text{disp}_1, \text{disp}_2, \text{disp}_3, \text{disp}_4 \);

\( NCC \) owns port \( nccP \) which is unrolled as \( nccP_1, nccP_2 \), and port \( nccPP \) made explicit as \( nccPP_1 \). Then, the transformation generates connectors for \( \text{disp}_\text{node} \) and \( \text{disp}_\text{port} \). For \( \text{disp}_\text{node} = \langle \text{assembly}, \langle \text{disp}, \text{dispP} \rangle, \langle \text{node}, \text{nodeP} \rangle \rangle \), the \textit{Redundancy} stereotype is applied and explicit connectors are generated according to the rules specified for the \textbf{redundancy pattern}.

Resulting connectors \( \text{disp}_\text{node}_i\text{conn} = \langle \text{assembly}, \langle \text{disp}_i\text{disp}, \text{dispP}_i\text{dispP} \rangle, \langle \text{node}_i\text{node}, \text{nodeP}_i\text{nodeP} \rangle \rangle \) are depicted in the left-hand side of Figure 12. Regarding \( \text{disp}_\text{port} = \langle \text{assembly}, \langle \text{disp}, \text{dispPP} \rangle, \langle \text{portH}, \text{portP} \rangle \rangle \), the \textit{MpartsNports} stereotype is applied and explicit connectors are generated according to the rules specified for the \textbf{M-parts–N-ports pattern}.

While previously we showed how the pattern is applied to \textit{delegation} cases, here we depict the application to an \textit{assembly} case. The resulting connectors \( \text{disp}_\text{port}_i\text{conn} = \langle \text{assembly}, \langle \text{disp}_i\text{disp}, \text{dispPP}_i\text{dispPP} \rangle, \langle \text{portH}_i\text{portH}, \text{portP}_i\text{portP} \rangle \rangle \) are shown in the right-hand side of Figure 12.

As described in the definition of connection patterns and semantic rules, it is important that the connector ends are ordered according to the rules defined for the specific connection pattern. Since in the modelling environment they are ordered according to their creation order, the M2M transformation takes care of re-ordering them when needed. Moreover, as mentioned before, the algorithm runs a validity check on the multiplicities of the actors of each connector to ensure that their combination obeys to the condition in Equation 1. This check is implemented in the transformation as an \textbf{assert fatal} that immediately stops the execution of the transformation if the condition is violated and informs the developer about the connector that violates the condition.

While the application of the transformation to this simplified model was meant to practically describe when and how the different connection patterns and semantic rules are applied, a more thorough evaluation has been performed on real industrial models of various sizes and using different input and output formats as described in Section 7.
7. Evaluation and Discussion

The presented solution has been evaluated on case-studies defined using plain UML as well as UML profiles (e.g., CHESS-ML [26]). The AAL2 subsystem models, defined within Ericsson Nikola Tesla in Zagreb (Croatia) under the supervision of Ericsson AB in Kista (Sweden), on which the solution was applied consisted of a maximum of 3000 component instances and 15000 port instances decomposed in a maximum of 550 hierarchical composition levels [27]. The calculation rules are defined a-priori and applied to one connector at a time, therefore increasing complexity of the composite structures does not jeopardise the accomplishment of the calculation but rather the scalability of the algorithm.

In this direction, despite the inclusion of more complex connection cases as well as the entailment of multiple connections, the identification of connection patterns together with re-ordering of connection ends as well as the complete re-definition of the semantic rules and the relaxation of the constraints on port types allowed this solution to be more scalable than what preliminarily presented in [7]. More specifically, for AAL2 subsystem models of the size mentioned above (i.e., $10^4$ component and port instances) the generation time was drastically reduced from $\approx 8$ to $\approx 3$ minutes on a Windows 7 64-bit machine running a 2.6GHz CPU and 8GB RAM. This performance improvement was achievable thanks to the introduction of the ECP profile that simplifies the identification of connection patterns and the rules to be applied. In order to reproduce similar models, it would be enough to add hierarchical levels as well as increasing multiplicities to the model elements shown in Section [6].

Figure 12: $disp_{-}node_{-}i_{conn}$ and $disp_{-}port_{-}i_{conn}$
In any case, the calculation itself can be carried out faster if no generation of physical artefacts (UML model in this case) is needed. We implemented Algorithm 1 in terms of a model-to-text transformation in Xtend\(^5\) for generating explicit interconnections in a textual format, in a similar way as expressed in Table 1. Run on the same models, the textual generation takes only few seconds to complete its task. This could be a solution to go for in case of very large systems towards code generation since, from a rendering point of view, it would be much less resource demanding than a UML model. In addition, it would be easier for the developer to grasp the interconnections at a glance without having to be aware of all the details related to the involved UML specific concepts.

In the scope of the SMARTCore project (see Acknowledgements), this solution permits to provide the generation of 100\% C++ code from design models for deployment and task allocation issues and optimisation\(^28\). The calculation and generation of explicit interconnections has in fact enabled the possibility to compute end-to-end response times and resource usages on chains of tasks derived from interconnections among component instances.

The syntactical expressive power (and to a certain extent semantics) of the UML metamodel, and in our specific case its definition of composite structures, enables the design of complex component-based systems. Nevertheless, in order to keep its general-purpose nature intact, a certain degree of variability in its semantics has been intentionally left open to different interpretations through the concept of semantic variation point. The main goal with that was for UML to be able to represent a family of languages with commonalities as well as variabilities that could be tailored to a given application domain or specific problem.

The specific case of setting semantic rules for matching multiplicities of interconnected parts and ports, while trivial in common examples used when describing “academic” systems (usually with multiplicities set as \(== 1\)), represents a serious challenge in the development of model-driven tools and techniques for large scale industrial systems. These often consist of several hundreds (even thousands) component and port instances from which, in order to be able to deterministically provide automation for model-based analysis, model simulation and execution as well as code generation, the importance of having a precise interconnection semantics cannot be disregarded. Thanks to the presented solution, we have been able to go from a partial\(^29\) to a fully fledged\(^30\) approach for preservation of extra-functional properties from models to code, entailing full code generation, and monitoring properties at instance level\(^31\).

In this work we provided a possible solution to this problem for a specific set of connection patterns. More specifically, we entail those cases where multiplicities are defined as concise values since for range values it would not be possible to completely automate the calculation and generation process without the user intervention. This is due to the implicit variability of the single actors (multiplicities of parts and ports) of the interconnections. Anyhow, it would certainly be interesting to explore these cases too, for instance by providing a semi-automated mechanism that would reduce the effort needed from the user side to a minimum.

\(^5\) http://eclipse.org/xtend/
In the same direction, since a semantic variation point allows different interpretations of a same variability point, a useful enhancement of the generation mechanism could be to introduce the concept of configurable profile and connection patterns. This could, for instance, allow the developer to (i) customise the semantics specified for a specific connection pattern, (ii) add alternative semantic rules for a same connection pattern depending on specific modelled patterns (e.g., as provided in this work for redundancy), and (iii) include new connection patterns as stereotypes in the profile with related semantic rules for generation.

8. Conclusion

UML composite structures represent a common way to model complex systems according to the component-based design pattern defined in the UML specification. When employing this formalism, a multiplicity can be specified for some of the involved modelling elements (e.g., parts, ports, connector ends), while connectors are not equipped with a detailed specification of the instances they connect.

On the one hand, a detailed specification of interconnections can be crucial for several activities, as the accurate analysis of important system properties at modelling level, model simulation, or the generation of full target code, and therefore it should be somehow included either at model or at code level. On the other hand, due to the complexity of modern systems, the effort that such an activity would require if carried out manually can easily become massive. Apart from excessive effort, such a manual activity, when performed at code level, may induce other discouraging drawbacks, such as unintentional injection of errors as well as inconsistencies between models and code.

To tackle this issue, in this article we provided a support for the automatic calculation and generation of explicit interconnections between instances according to their respective multiplicity. This support is achieved by exploiting the semantic variation points mechanism provided along with UML, and specifically by: providing (i) a set of five connection patterns and related semantic rules to give a semantics to the interconnections among instances, (ii) the ECP profile that enables modelling and validation of connections patterns for UML models, and (iii) a generation algorithm. The latter exploits connection patterns and semantic rules to generate, from a component-based UML model in input, a new version of it obtained by unrolling parts and ports as well as by generating explicit interconnections among the unrolling elements. The algorithm is meant to be independent of both its output format and the technology used for implementing it.

In order to provide a proof-of-concept of the generation of explicit interconnections we implemented the generation algorithm as an M2M transformation defined using Operational QVT and embedded in a Eclipse plugin as an extension of the Papyrus modelling environment. The proposed solution has been evaluated on industrial models of various sizes in collaboration with Ericsson, starting from both plain UML and UML profiles and generating different types of output artefacts.
Acknowledgements

The research work presented in this article has been supported by the Knowledge Foundation through the SMARTCore (Model-driven support for deployment optimization of embedded systems on multicore) project and the following industrial partners: Ericsson AB, ABB Corporate Research and Alten AB Sweden.


URL http://dx.doi.org/10.1007/978-3-540-87875-9_57


Appendix A. Composite Structure Concepts

**Definition 1.** A component represents a modular constituent of the system which encapsulates its internal environment and whose manifestation is replaceable within its environment [6]. We formalise a component as a tuple $C = \langle pr\{\}, p\{\}, con\{\} \rangle$, where $pr\{\}$ represents a set of owned parts, $p\{\}$ represents a set of owned ports, and $con\{\}$ represents a set of owned connectors. In Figure A.13, Component represents $C$ and contains (i) parts, representing $pr\{\}$, (ii) ports, representing $p\{\}$, and (iii) connectors representing $con\{\}$.

**Definition 2.** A part is a role played by a set of instances of a component at runtime; a part can include a multiplicity factor [6]. We formalise a part as a pair $pr = \langle C, M_{pr} \rangle$, where $C$ is a component and represents the part’s type, and $M_{pr}$ represents the part’s concise multiplicity, with $M_{pr} \geq 1$. In Figure A.13 Part represents $pr$ and it points to its type $C$ through the relation ofType. Moreover, multiplicity represents $M_{pr}$. Note that the same part cannot belong to several composite structures.

**Definition 3.** An explicit part is a role played at runtime by exactly one instance of a component. We introduce this element in order to be able to unambiguously represent explicit interconnections among parts representing sets of instances when $M_{pr} > 1$. An explicit part is formalised as $pr_{ipr} = \langle C, 1 \rangle$ where $ipr \in [1, M_{pr}]$.

**Definition 4.** A port is an interaction point which is used to connect structured components with their parts as well as with the surrounding environment and it may contain a multiplicity factor representing the number of port instances at runtime [6]. We
formalise a port as a pair \( p = \langle I, M_p \rangle \), where \( I \) represents the ports type (i.e., an interface) and \( M_p \) represents the port’s concise multiplicity, with \( M_p \geq 1 \). In Figure A.13, \( \text{Port} \) represents \( p \) and it points to its type \( I \) through the relation \( \text{ofType} \). Moreover, \( \text{multiplicity} \) represents \( M_p \).

**Definition 5.** An explicit port represents a single port instance at runtime and it is introduced for representing explicit interconnections when a port \( p \) has \( M_p > 1 \). This element is formalised as \( p_i.p = \langle I, 1 \rangle \) where \( i_p \in [I, M_p] \).

**Definition 6.** A connector end is an endpoint of a connector, which attaches the connector to a connectable element (i.e., port) [6]. This element is formalised as a pair \( ce = \langle pr, p \rangle \), where \( pr \) represents a part and \( p \) represents the part’s port (addressed also as role) to which the connector is connected. In case of a delegation (see Definition 7) connector, the connector end related to the delegating component has empty \( pr \) (represented as \( \text{null} \)). In Figure A.13 the element \( \text{ConnectorEnd} \) represents \( ce \) and points to \( \text{ConnectableElement} \) specialised by \( \text{Port} \) that represents \( p \). The part \( p \) is identified through the relation \( \text{part} \).

**Definition 7.** Connections between ports are achieved through connectors, which can be of two kinds: delegation and assembly. A delegation connector represents the forwarding of events meaning that what arrives at a port that has a delegation connector to one or more parts or ports on parts will be passed on to those targets for handling. An assembly connector links two or more parts or ports on parts (at the same hierarchical level) that defines that one or more parts provide the services that other parts use. A connector is formalised as a tuple \( \text{conn} = \langle T, ce_1, ce_2 \rangle \), where \( T \in \text{assembly, delegation} \) represents the connector’s type and the two connector ends are represented by \( ce_1 \) and \( ce_2 \). In Figure A.13, \( \text{Connector} \) represents \( \text{conn} \) and it contains the two connector ends \( ce_1 \) and \( ce_2 \) that it connects through the composition \( \text{ends} \). Note that the type of the connector is not explicit in Figure A.13 but rather derived from the connector ends according to Definition 6.

**Definition 8.** An explicit connector is defined as a tuple \( \text{conn}_j = \langle T, \langle pr_1p, p_1p \rangle, \langle pr_2p, p_2p \rangle \rangle \), where \( j \in [1, M_{pr} \times M_p] \) and represents one element in the set of explicit interconnections represented by \( \text{conn} \).

Note that the concepts we introduced, namely explicit part, explicit port and explicit connector, are not explicitly visible in the composite structure metamodel in Figure A.13 but they are part of the solution for fixing the semantic variation point.
Appendix B. OCL Constraints for Semantic Checking of Multiplicities

The constraint expression in Listing 1 is defined in the context of the stereotype `ExplicitConnector` and checks, according to Equation 1, the general conditions for any of the stereotypes to be applied. More specifically, it checks that the connector has exactly two connector ends (line 1), and that, if the connected parts are not empty (lines 3-5), then $M_a \times M_{ap} == M_b \times M_{bp}$ (lines 7-10). Alternatively, if part $a$ is empty (line 12) then $M_{ap} == M_b \times M_{bp}$ (lines 14-16), or, if part $b$ is empty (line 18) then $M_a \times M_{ap} == M_{bp}$ (lines 20-22).

Listing 1: CheckMainRule

```ocl
def self.base_Connector._end->size() = 2
and
((self.base_Connector._end->first().partWithPort<>null)
and
(self.base_Connector._end->last().partWithPort<>null)
and
self.base_Connector._end->first().role.oclAsType(Port).upper=
self.base_Connector._end->last().partWithPort.upper*
self.base_Connector._end->last().role.oclAsType(Port).upper))
or
((self.base_Connector._end->first().partWithPort=null)
and
(self.base_Connector._end->first().role.oclAsType(Port).upper=
self.base_Connector._end->last().partWithPort.upper*
self.base_Connector._end->last().role.oclAsType(Port).upper))
or
((self.base_Connector._end->last().partWithPort=null)
and
(self.base_Connector._end->last().role.oclAsType(Port).upper=
self.base_Connector._end->first().partWithPort.upper*
self.base_Connector._end->first().role.oclAsType(Port).upper))
```

The constraint expression in Listing 2 checks the condition ($M_a == M_b$, and thereby $M_{ap} == M_{bp}$ as checked by `CheckMainRule`) for applying the `Array` stereotype and it is defined in the context of `Array`.

Listing 2: CheckArrayMult

```ocl
(self.base_Connector._end->first().partWithPort.upper=
self.base_Connector._end->last().partWithPort.upper)
```

The constraint expression in Listing 3 checks the condition for applying the `Redundancy` stereotype and it is defined in the context of `Redundancy`. More specifically it checks that $M_a == M_{ap} == n$ (and therefore $M_b == M_{bp} == m$, lines 1-2), that $n > 1$ (by checking that $(M_a \text{ mod } 2) == 0$, line 4), and $m > 2$ (line 6).

Listing 3: CheckRedundancyMult

```ocl
(self.base_Connector._end->first().partWithPort.upper=
self.base_Connector._end->last().role.oclAsType(Port).upper)
and
(self.base_Connector._end->first().partWithPort.upper>1)
```
The constraint expression in Listing 4 checks the condition for applying the Star stereotype and it is defined in the context of Star. More specifically it checks that $M_a == M_b == 2n$ (line 1-2) and that $n \geq 1$ (line 4). Moreover, it checks that $M_{ap} == 2$ (and $M_{bp} == 2$, line 6) and that $(size \times 2) \leq M_a$ as well as $M_a$ is divisible by $(size \times 2)$ (lines 8-9).

Listing 4: CheckStarMult

```python
1 (self.base_Connector._end->first().partWithPort.upper ==
2 self.base_Connector._end->last().partWithPort.upper)
3 and
4 (self.base_Connector._end->first().partWithPort.upper.mod(2)==0)
5 and
6 (self.base_Connector._end->first().role.oclAsType(Port).upper==2))
7 and
8 (self.base_Connector._end->first().partWithPort.upper.mod(
9 self.size*2)==0)
```

The constraint expression in Listing 5 checks the condition for applying the Perfect-Shuffle stereotype and it is defined in the context of PerfectShuffle. More specifically it checks that $M_a == M_b == 2n$ (line 1-2) and that $n > 2$ (line 4). Moreover, it checks that $M_a > 2$ (therefore $M_b > 2$, line 4), that $M_a$ (therefore $M_b$) is even (line 6) and $M_a == 2$ (therefore $M_b == 2$, line 8).

Listing 5: CheckPerfectShuffleMult

```python
1 (self.base_Connector._end->first().partWithPort.upper ==
2 self.base_Connector._end->last().partWithPort.upper)
3 and
4 (self.base_Connector._end->first().partWithPort.upper>2)
5 and
6 (self.base_Connector._end->first().partWithPort.upper.mod(2)==0)
7 and
8 (self.base_Connector._end->first().role.oclAsType(Port).upper==2))
```