Exploiting UML Semantic Variation Points to Generate Explicit Component Interconnections in Complex Systems

Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin
Mälardalen University (MRTC) - Västerås, Sweden
email: [federico.ciccozzi, antonio.cicchetti, mikael.sjodin]@mdh.se

Abstract—Modern modelling languages provide dedicated features to support the detailed design of complex systems from different domains. Nevertheless, especially general-purpose languages provide extensive syntactical expressiveness that, in some cases, may be hard to align with well-defined semantics. This is often inevitable due to the fact that, in order for model-driven approaches to be fruitful, modelling activities should require a lower effort than code-based approaches. In the case of UML and component-based design, precise syntactical definition of the number of component and port instances building up the system is provided. How component instances are explicitly connected to each other via ports is left as a semantic variation point in order for the modeller to freely interpret the related metamodel semantics. On the one hand, in order to allow model analysability, simulation and generation of target code, ad-hoc choices regarding this semantics need to be taken. On the other hand, leaving the burden of this task to the developers is often unthinkable for complex industrial systems (composed by a number of components and connections in the magnitude of $10^4$ and above). Therefore we propose a set of semantic rules for the establishment of explicit links between component instances and provide a solution for the automatic generation of these links by applying the defined rules.

Keywords—model-driven engineering; component-based software engineering; model transformations; explicit instance model; semantic variation point;

I. INTRODUCTION

Model-Driven Engineering (MDE) proposes to face the increasing complexity of modern software systems by shifting the focus of the development from coding to modelling. Models abstract the reality by providing only those details that matter for the particular problem taken into account and are built by following a set of rules prescribed by means of a language definition, referred to as metamodel [1]. In order for MDE to succeed, it is of paramount importance that modelling tasks are proven to be more efficient than coding; in fact, if the complexity of the former was comparable with the latter’s, the return of investment of such a technology shift would not be encouraging [7]. This aspect is especially relevant in industrial settings, where systems’ size and lifespan push development approaches to their efficiency limits.

In the scope of MDE applied to industrial development, the UML (Unified Modeling Language) has been widely adopted and many tool vendors produce useful tools, some of which reach a certain degree of industrial acceptance. At the same time UML’s fuzziness has made it difficult to build complete code generators, simulators, model-based analysis or testing tools working in a consistent manner. Many of these fuzziness issues have been tough solved with UML 2.0, partly thanks to the notion of semantic variation points. 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 the interpretation of the metamodel semantics. Thanks to semantic variation points, UML becomes a family of languages with commonalities as well as variabilities that can be tailored to a given application domain [3].

This paper exploits the notion of semantic variation point offered by the UML to provide a possible interpretation of the UML metamodel semantics for the establishment of explicit interconnections between software components through ports ("The rules for matching the multiplicities of connector ends and those of parts and ports they interconnect are a semantic variation point.", in OMG UML Superstructure [10]). In particular, UML enables the specification of arbitrary, in terms of multiplicity, relationships among components through their ports [10]. However, for analysis purposes or code generation it is necessary to exactly know how those interconnections will be eventually realised at instance level.

In this work we provide a set of semantic rules for the establishment of explicit links between component instances. Note that, when referring to an element (i.e., component, port) instance, we mean a single entity in the related classifier’s multiplicity range and therefore not to be confused with its instantiation at code level. A set of assumptions, that will be explained in a dedicated section, have been made to clarify the scope of both problem and solution. Moreover, since manual modelling of these links is impracticable in most of industrial cases (due to their size and complexity), we also propose an automated approach that is able to generate the instantiation of both component instances and explicit links according to the given semantic rules. The approach is validated against industrial case-studies; throughout paper we discuss the outcomes of this validation.

The remainder of the paper is structured as follows. In Section II we provide the identification of the context in which we define problem and intended contribution as well as the current state-of-the-art. The addressed problem is formally defined in terms of motivation, challenges and assumptions in
Section III. The description of the proposed solution is fully unwound in Section IV, while its application to a sample case-study is shown in Section V. Additional results derived from the validation of the approach against industrial case-studies is presented in Section VI. The paper is concluded by Section VII where a summary of the presented work and intended future enhancements of the proposed approach are given.

II. BACKGROUND AND RELATED WORK

In this paper we exploit the CHESS modelling language (CHESS-ML) [4], defined within the CHESS project (cf. Acknowledgements) as a set of UML profiles and employing the component-based design as reference modelling pattern. Thanks to the proposed solution, in the scope of CHESS we have been able to provide predictability analysis of system models as well as the generation of 100% of target code (e.g., C++ for telecom systems) together with propagation of code analysis results back to system models [5].

By component-based design (or modelling) pattern we mean designing the system under development as assembly of components (see Section III-B). The concept of component has been introduced in the UML 2.0, together with an appropriate diagram [10]. In [2] the author discusses composition mechanisms provided with UML 2.0, and in particular the role of multiplicities in interconnections between composite structures. However, it only touches upon the issue of realising component interconnections at different abstraction levels while leaving aside the concrete instantiation.

Several research works have been devoted to giving semantics across links through ports and keeping its correctness [6], [9], [11], nonetheless they focus more on interface definitions (in terms of type and behaviour of ports and connectors) rather than on the issue of explicitly instantiating instance-to-instance links between components. Even component-based design tool implementations, as the solution proposed in [13], seem to miss out the problem we are addressing in this paper.

Concerning the tools providing code generation from UML models, different solutions are provided when coming to the generation of interconnections between component instances. Enterprise Architect 1, by Sparx Systems, provides code generation from class diagrams where classes are linked through associations. More specifically, code is generated so that instances of the association’s target class are owned by the instances of the source class. In our solution we target component-based development and aim at generating code which preserves this paradigm, that is to say components communicating by invoking functionalities on their own required ports with no need of knowing which component is on the other side of the connection providing the functionality. 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 [12].

Other solutions, as in IBM Rational Rhapsody 2, maintain the generality of UML when coming to matching the multiplicities of components and ports. In fact, no decision is automated regarding the interconnections between components via ports, but rather instances generated according to their multiplicities along with function handlers (i.e., get and set) for managing connections when needed. In this case, the modeller will have to specify how to connect the different component instances when describing the behaviour of the single components. Our approach provides an interpretation of the UML metamodel’s semantics, as prescribed by the notion of semantic variation point [10], in order to automate the generation of fixed interconnections between component instances whose manual specification would require heavy and error-prone modelling effort in case of complex industrial systems composed by several hundred thousands components. That is to say that, despite analysis, simulation, and code generation techniques remain valid without any form of components’ instantiation automation, such a problem can remarkably affect scalability if not automated.

III. PROBLEM FORMALISATION

In this section the problem is formalised in terms of motivation and intended contribution, identified core challenges to be solved in order to achieve the final goal and assumptions made to bound the scope of problem and solution.

A. Motivation

When defining the system in terms of components and ports, for each of these elements minimum and maximum allowed or expected amount (i.e., multiplicity) of instances can be specified. As depicted on the left-hand side in Fig. 1, the two components (or sets of instances since their multiplicity is greater than 1) $A$ and $B$ are linked through a single connector that does not entail any information on the actual instance-to-instance connection. This information is crucial to be able to properly analyse important properties of the system at modelling level, enable model simulation and generate executable target code. That is why semantic rules should be specified for generating the actual connections among explicit component instances. This information could be manually modelled in different ways (e.g., ad-hoc stereotyped annotations, OCL constraints) or created on-the-fly (e.g., through action languages), but, when dealing with complex systems consisting of thousands of component instances, the manual effort demanded by such an activity is overwhelming. Alternatively, the same information could even be manually described at code level; different complications, such as inconsistency between models and code, unintentional injection of errors in the code as well as large effort in carrying out the manual coding task arise thus making this solution simply infeasible in developing industrial systems. For these reasons, automation is needed in order to generate it and generally mitigating the developing team’s effort demanded by modelling activities.

The outcome of this work is a solution for the automatic generation of explicit component instances and establishment of links among them according to the involved components’ and ports’ multiplicities, in the structural model of the system.
This generation is made possible by a set of semantic rules, presented in Section IV, that we defined as semantic interpretation of the UML metamodel by exploiting the semantic variation points mechanism provided along with UML. On the right-hand side in Fig. 1 the general idea of generating explicit component instances and links is applied to the two sets of instances A and B introduced before. More specifically, the components A and B are unwound by means of explicit instances according to their multiplicity and the actual links among them are created. Moreover, we provide validation of the proposed approach against industrial case-studies as discussed in Section VI.

**B. Challenges**

The UML allows the specification of a system as assembly of components communicating via required and provided interfaces (exposed by ports). As stated in the language superstructure specification, a port represents an interaction between a classifier instance and its internal or external environment. Additionally, features owned by required 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 [10]. In other words, multiple instances of components can provide features via ports to multiple instances of both peer (at the same hierarchical level) and internal components. While the number of instances of components and ports can be precisely specified, the port-to-port links are not equipped with a detailed specification of the component instances they connect. In fact, the links are defined at classifier level, thus on the sets of instances.

From this point, in order to be able to automatically generate the set of actual links between explicit component instances the solution cannot prescind from adding the semantic information needed to drive the definition of links’ source and target. This information entails the ability to derive explicit links between instances based on the multiplicities defined on components and ports. Moreover, UML allows the definition of hierarchical composite components with unlimited depth levels, therefore this hierarchical structuring must be taken into account. In fact, links do not always concern component instances placed at the same hierarchical level but even instances defined at different depths in the composition structure. Especially when composition hierarchy does not imply delegation of the provided features by internal classifiers to the container classifier, the ability to generate explicit links between component instances placed at different hierarchical levels becomes crucial for increasing model analysability and providing complete code generation capabilities. Eventually, the generated information has to be stored in structures compatible with the modelling environment in order to be used for those purposes.

To summarise, in order to provide a solution to the described problem we need to tackle the aforementioned challenges by: (i) defining semantic rules for driving the generation of actual links between explicit component instances via ports, (ii) identifying appropriate means for storing the generated information, (iii) and performing the actual generation following the defined rules as well as the hierarchical composition of components and ports.

**C. Assumptions**

In order to identify the scope in which our solution has been developed and its ultimate goal, a number of assumptions and clarifications must be made. According to the constraints given when defining the CHESS-ML, for ensuring guarantees at runtime of the properties modelled at design time, dynamic instantiation of components is not allowed; that is the reason for which our solution entails only prefixed cardinality on components and ports. For the same reason, multiplicities are defined as concise values (i.e., \([n..m]\)) while range values (e.g., \([n\ldots m]\)) are left as future enhancement. Moreover, two further assumptions have been made in regards to components interconnections: (i) the connectors linking components via ports have multiplicity 1, thus leaving components’ and ports’ multiplicity as variables for the interconnections calculation and generation, (ii) only binary connections are considered, leaving n-ary possibilities as future work.

In any case, these assumptions did not prevent us from the ability to model complex industrial systems, but were rather derived to circumscribe the scope of the problem and therefore propose a technically valid solution. Since no unique interpretation of the UML metamodel semantics can be given for a semantic variation point, we focused on the problems to be addressed within the CHESS project with particular attention to full-fledged code generation.

**IV. PROPOSED SOLUTION**

The final goal of the proposed approach is to generate an instance model containing all the information regarding explicit component and port instances as well as explicit links among them. Taking the structural definition of the system modelled using UML and the defined semantic rules as input, a generation routine creates the instance model in conformance to the defined instance metamodel. System models and instance model together provide the needed syntactical and semantic precision to be used as source artefacts for performing analysis, simulation and code generation. In Fig. 2 the proposed solution is shown; the artefacts defined for the
process are depicted in bold and represent the solutions for the challenges described above.

\[ M_A \times M_{A\_RP} = M_B \times M_{B\_PP} \]  

(1)

At this point, given both the component instance \( i_A \) and the instance \( i_{A\_RP} \) of its required port, we automatically derive the link to the right instance \( i_B \). We defined the following set of five semantic rules for creating such links based on the specified multiplicity combinations (where \( \lceil x \rceil \) denotes the ceiling function mapping \( x \) to the smallest integer greater than or equal to \( x \)):

\[ (M_A = M_B) \land (M_{A\_RP} = M_{B\_PP}) \Rightarrow i_B = i_A \quad \text{(Rule 1)} \]

\[ (M_A \neq M_B) \land (M_A = 1) \land (M_{A\_RP} = M_B \times M_{B\_PP}) \Rightarrow i_B = \left\lfloor \frac{i_{A\_RP}}{M_{B\_PP}} \right\rfloor \quad \text{(Rule 2)} \]

\[ (M_{A\_RP} \neq M_{B\_PP}) \land (M_{A\_RP} = 1) \land (M_A = M_B \times M_{B\_PP}) \Rightarrow i_B = \left\lfloor \frac{i_A}{M_{B\_PP}} \right\rfloor \quad \text{(Rule 3)} \]

\[ (M_A \neq M_B) \land (M_B = 1) \land (M_{B\_PP} = M_A \times M_{A\_RP}) \Rightarrow i_B = 1 \quad \text{(Rule 4)} \]

\[ (M_{A\_RP} \neq M_{B\_PP}) \land (M_{B\_PP} = 1) \land (M_B = M_A \times M_{A\_RP}) \Rightarrow i_B = \frac{(i_A \times M_{A\_RP}) - (M_{A\_RP} - i_{A\_RP})}{M_{B\_PP}} \quad \text{(Rule 5)} \]

The same rules can be applied in case of delegation among provided or required ports in order to generate links between them; in this case, \( A\_RP \) and \( B\_PP \) would both represent, respectively, provided or required port instances. Note that the defined rules are mutually exclusive therefore there is no fixed order in which they are supposed to be applied.

The considered set of multiplicity combinations does not cover all the possibilities that satisfy equation 1, but rather the set for which no added modelling effort was requested from the developer side. The possibility which is not covered is represented by the equation 2 and represents the case in which equation 1 is satisfied and none of the operands is equal to 1.

\[ (M_A = M_{B\_PP}) \land (M_B = M_{A\_RP}) \land (M_A \neq M_B) \quad \text{(2)} \]

Future developments targeting these extensions are introduced in Section VI.

**B. Definition of the Instance Metamodel**

The explicit instances of components and ports as well as the links among them need to be stored in properly defined structures used both during the generation process itself and eventual system model analysis, simulation or code generation. In an MDE development process, the most suitable
way to store information concerning model elements is to use models. Due to performance reasons arising from our specific application of the approach to code generation within CHESS, we defined an Instance Metamodel \((\text{InstanceMM})\) using Ecore\(^3\) for such purpose. In any case, the provided approach is independent of the adopted modelling language. In fact, for model analysis and simulation purposes, we stored the information regarding components and links instances by means of instance specifications available in UML. Let us generalise the representations introduced when defining the semantic rules by the concepts provided by InstanceMM. Considering component \(A\), the instances \(i_A\) are represented by the Component metaclass; the attributes \(\text{mult}\) and \(\text{rel}_id\) represent respectively the component multiplicity \(M_A\) and the value assumed by \(i_A\) in the range \([1, M_A]\). Mirror reasoning applies to component \(B\). Port instances \(i_{A\text{RP}}\) and \(i_{B\text{PP}}\) are represented by, respectively, RequiredPort and ProvidedPort that specialise the abstract metaclass Port. The multiplicities \(M_{A\text{RP}}\) and \(M_{B\text{PP}}\) are modelled through the Port’s attribute \(\text{mult}\). The value assumed by \(i_{A\text{RP}}\) in \([1, M_{A\text{RP}}]\) and by \(i_{B\text{PP}}\) in \([1, M_{B\text{PP}}]\) are represented by the Port’s attribute \(\text{id}\).

After generating explicit instances of components and ports in the terms described above, it is possible to apply the defined semantic rules for creating the explicit links between component instances. As aforementioned, given \(i_A\) and \(i_{A\text{RP}}\), the idea is to find the right instance \(i_B\) to which \(i_A\) is connected through \(i_{A\text{RP}}\); once found, this index is stored in the attribute \(\text{dest}_id\) (or \(\text{deep}_id\) if considering provided ports) of \(i_{A\text{RP}}\), thus making \(i_{A\text{RP}}\) point to the right instance \(i_B\). In this way, at the end of the process, we will obtain an instance model conforming to \(\text{InstanceMM}\) and containing all the explicit instances of components and ports as well as the actual instance-to-instance links.

C. Generation Process

The generation of explicit instances and links is achieved through a model-to-model transformation defined in Operational QVT\(^4\). Taking as input a component-based description of the system defined in UML, the transformation generates the explicit instances of components and ports and then creates the explicit links between component instances by applying the previously defined semantic rules. The multiplicities must be aligned to the rules defined earlier in this section for the generation process to operate correctly. The output of the transformation is an instance model which conforms to \(\text{InstanceMM}\).

The transformation has to take into account all the possible connections between ports in a component-based design pattern with unlimited hierarchical levels. These connections can be summarised as follows:

- **Provided to Provided**: in case of composite structures, container and contained component instances can be connected via provided ports for modelling delegation of features’ provision visibility to the environment.
- **Required to Required**: similarly, container and contained component instances can be connected via required ports for modelling delegation of features’ request to the environment.
- **Required to Provided**: connecting component instances via a link between required and provided port respectively, represents the actual client-server interaction where a component instance owning the required port requires features that the one owning the provided port offers.

The transformation workflow is summarised in the pseudocode shown in Algorithm 1.

Algorithm 1 M2M Transformation from UML model to instance model

\[
\text{Uml2Instance}(\text{in UmlM, out InstanceM})
\]

\[
\text{for each component } c \text{ in UmlM do}
\]

\[
\text{InstanceM} = c.\text{createInstances}();
\]

\[
\text{end for}
\]

\[
\text{for each comp instance } c\text{Inst in InstanceM do}
\]

\[
c\text{Inst settProv2Prov}();
\]

\[
\text{end for}
\]

\[
\text{for each comp instance } c\text{Inst in InstanceM do}
\]

\[
c\text{Inst settReq2Prov}();
\]

\[
\text{end for}
\]

\[
\text{for each comp instance } c\text{Inst in InstanceM do}
\]

\[
c\text{Inst settReq2Req}();
\]

\[
\text{end for}
\]

The main transformation rules work as follows:

- **createInstances()**: for each component in the UML model (UmlM), a set of Component elements is created in the instance model (InstanceM); in addition, ProvidedPort and RequiredPort elements are created for both provided and required ports of the UML component. The number of component and port instances to be created is represented by the upperBound multiplicity of the related UML element. Moreover, the hierarchical structure of the components is kept intact in order to correctly generate the links between them.
- **setProv2Prov()**: in the UML model, container components may be connected to contained components via provided-to-provided port connection. In the instance model, for each generated component instance, starting from the root\(^5\), we create the explicit links between its provided ports to the component instance owning the provided port at the other end of the connection. The rule is then recursively applied to the contained component instances.
- **setReq2Prov()**: in the UML model, peer components are connected via required-to-provided port connection.

---

\(^3\)http://www.eclipse.org/modeling/emf/?project=emf

\(^4\)http://www.eclipse.org/projects/project.php?id=modeling.m2m.qvt-xml

\(^5\)By root component it is meant the one at the root of the hierarchical composition. For peer components we mean components placed at the same hierarchical level.
In the instance model, for each generated component instance, starting from the root, we create the explicit links between its required ports to the component instance owning the provided port at the other end of the connection. The rule is then recursively applied to the contained component instances.

- **setReq2Req():** at this point, each explicit required port instance points to the right component instance owning the provided port instance at the other end of the connection. In the UML model, container components may be connected to contained components via required-to-required port connections. In this case, the transformation sets these missing links in a similar way as for provided ports in **setProv2Prov().** The rule is then recursively applied to the contained component instances.

Setting links between port instances, regardless of the connection type (i.e., required to provided, provided to provided, required to required), is done by applying the semantic rules previously defined.

### V. A Case Study: The AAL2 Subsystem

The solution proposed in this work has been validated against industrial case-studies modelled in CHESS-ML as described in Section VI. In order to show a concrete application of the proposed solution, we employ the Asynchronous Transfer Mode (ATM) Adaptation Layer 2 (AAL2) subsystem, originally intended to adapt voice for transmission over ATM and currently used in telecommunications as part of connectivity platform systems. The AAL2 subsystem is composed by several hundred thousands of component instances and multiple levels of hierarchical composition of components.

In Fig. 3 we propose a simplified version of the AAL2 subsystem (i.e., SwSystem composite component) 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 example we consider only part of the NCC internal structure while considering AAL2RI_Client and NCIClient as stubbed. NCC is a connections handler providing connectivity services for the establishment/release of communication paths between pairs of connection endpoints handled by AAL2RI_Client. NCIClient represents an application asking for services provided by NCC and its underlying layers; the components communicate through functional interfaces (function calls or message passing depending on the deployment configuration) exposed by their provided ports.

The NCC component has a complex internal structure (Fig. 3), and in this study we focus on: NodeConnHandler, which dispatches the incoming connection requests to available NetConn instances, NetConn, that controls establishment and release of network connections between nodes (NodeConnElem instances), NodeConnElem, 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 case-study we consider only the first two levels of decomposition (down to the NCC’s internal structure).

The behavioural definition of the system (NodeConnHandler state-machine in Fig. 4) is given by means of state-machines enriched with action code definitions for the involved operations specified by means of the Action Language for Foundational UML (ALF). Components are connected by means of ports and links between them. The communication is thereby performed by calling operations on the component’s required ports that propagate the invocation to the component owning the provided ports connected to them (note that connected provided and required ports share the same Interface).

A typical connection scenario in the AAL2 subsystem is the establishment of a connection between two end-points residing on the same node. This is a constrained case of a more general network-wide connection where the two end-points reside

---

6http://www.omg.org/spec/ALF/
Fig. 4. NodeConnHandler state-machine in CHESS

Fig. 5. UML-like Instance Model for the AAL2 subsystem

on different nodes and the communication transits through a number of other intermediate nodes in the network. When NCIClient wants to connect two end-points, a connection setup request is sent to NCC through the PI_NCI_2_NCC interface; such request contains information about the end-points. 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 through the RI_NCC_2_AAL2RI interface for each end-point to their respective AAL2RI_Client to activate the access to the transport layer. Once both end-points have positively responded through their respective RI_AAL2RI_2_NCC interface, NCC confirms the establishment of the connection to NCIClient through the RI_NCC_2_NCI interface. In Fig. 4, the state-machine describing the behaviour of NodeConnHandler component is shown. ALF code specifying the behaviour of the component’s operation sendResponse (matching the homonymous state-machine’s transition) is also shown in the figure. Communication between components in terms of operations (node2clientResponseOk()) called on required ports (RI_NetDisp_2_NCC[index]) is depicted in the action code fragment. By calling a function on the port RI_NetDisp_2_NCC[index], the component NodeConnHandler is identifying exactly which of the NciClient components (there are 4 instances in the system) components to send the request to. Here is a typical scenario in which, without a precise semantics univocally defining the interconnections between components, it would not be possible to fully automate model-based analysis nor full-fledged code generation. After applying our generation process the instance model obtained as result contains explicit instances of components and ports as well as the actual links between component instances. Since the instance model is kept on a textual format for avoiding graphical rendering issues when dealing with complex systems, a UML-like graphical version has been created for this example and depicted in Fig. 5. According to the multiplicities defined in the CHESS model and following the semantic rules defined in the previous section, the transformation generates the following component entities within the SwSystem composite component: four instances of NciClient, one of AAL2RI_Client and one of NCC in turn composed by one instance of NodeConnHandler, four of NetConn, eight of NodeElem and four of NodeConnHandler.

Moreover, for each of them, instances for related required and provided ports have also been generated as well as the links among them (as shown in Fig. 5). Even though being a very simplified version of the actual AAL2 subsystem, the example proposed already highlights the complexity of setting interconnections among component instances; in fact, from a system model composed by 7 components, 24 ports and 14 connectors, the transformation mechanism generated an instance model composed by 23 component instances, 136 ports and 80 connectors. While the automatic generation of these instances took only few seconds, a manual modelling of each single interconnection instance, besides being error-prone, would not have been as fast; these advantages are amplified when dealing with more complex models as described in Section VI.

VI. VALIDATION

The presented approach has been developed within the CHESS project and validated against industrial case-studies coming from different application domains. The modelled
systems were composed by up to 2003 component instances, 14000 port instances and their hierarchical composition reached a maximum depth of 5 levels.

Validating the approach against complex case-studies gave us the possibility to evaluate scalability characteristics of the proposed solution. Within the AAL2 case-study we tested several design model sizes in order to thoroughly evaluate possible consequences in terms of scalability. The proposed solution resulted very scalable up to \(10^4\) component and port instances (i.e., within 8 minutes on machine running on a 2.6GHz CPU and 8GB RAM) while degrading going toward \(10^5\) (i.e., over 20 minutes on the same machine); in any case the process always accomplished its goals. This degradation can be explained by the complexity of the involved computations. More specifically, given \(n\) as the number of total component instances and \(m\) as the number of total port instances, the general limit behaviour of the computation represented by the formula \(3n \times m\) can be defined as \(O(n^2)\) since \(m \approx n\).

In this work we focused on binary connections between components; the transformation process would face exponential growth in its execution time in the case of n-ary connections. The consideration of these more complex cases have not been explored yet, since considered out of scope in the context of the CHESS project. Nevertheless, in order to widen the applicability of the approach, the semantic rules and the transformation process will be enhanced to consider n-ary connections as well as range values (i.e., \([n..m]\)) for multiplicity specification.

Within CHESS project, thanks to this solution we have been able to provide predictability analysis of system models for different industrial domains as well as the generation of 100% C++ code targeted to telecommunications applications. The process of code generation, which transparently applies the proposed approach to generate full-fledged functional code from CHESS models, and the execution of the generated code have been extensively validated in industrial settings [8] at Ericsson Nikola Tesla, Croatia.

VII. CONCLUSION

Modern modelling languages provide powerful features for dealing with systems’ ever increasing complexity sometimes leaving an intentional degree of freedom in the interpretation of their semantics. In this paper we exploit the notion of semantic variation point offered by the UML to provide a possible interpretation of the UML metamodel semantics for the establishment of explicit interconnections between software components through ports for which we proposed a set of specific semantic rules. Moreover, a solution for the automatic generation of the instances according to the proposed semantic rules is provided for alleviating the developers’ modelling effort when dealing with complex systems. In order to achieve this solution, the following challenges had to be tackled: (i) definition of semantic rules for driving the generation of actual links between explicit component and port instances, (ii) automatic generation of instances and links according to the defined semantics, and (iii) identification of appropriate structures for storing the generated information. The proposed approach has been developed within the CHESS project and validated against a number of industrial case-studies from different applicable domains.

The set of multiplicity combinations considered in the proposed semantic rules does not cover all the possibilities that satisfy the basic condition; future enhancements of the approach will target extensions in such sense, widening the allowed set of combinations towards a full alignment between syntactical and semantic expressiveness of the considered component-based modelling constructs.

Moreover, possible optimizations of the generation algorithm may be explored in order to enhance its scalability. Besides a correct UML system model, the algorithm does not need any other input from the developer for generating the instance-to-instance links between components. Possible future works could comprise, for instance, the possibility for the developer to customize such generation by proper annotation of the links at modelling level for manually explicitation of source and target instances. While for small systems this could even replace the defined semantic rules, in case of complex systems, it could only be seen as a way for specifying particular cases that would evade from the semantic rules which would still drive the automatic generation.

ACKNOWLEDGEMENTS

This research work was carried out within the CHESS project (ARTEMIS JU grant nr.216682, http://chess-project.ning.com).

REFERENCES