Lightweight Realization of UML Ports for Safety-Critical Real-Time Embedded Software

Alper Tolga Kocataş[1], Mustafa Can[1] and Ali Hikmet Doğru[2]
[1]*Avionics Software Design Department, Aselsan Inc., Ankara, Turkey*
[2]*Department of Computer Engineering, Middle East Technical University, Ankara, Turkey*

Keywords: UML, Model, Port, Object-Oriented, Realization, Transformation, Safety-Critical, Embedded, Real-Time.

Abstract: UML ports are widely used in the modeling of real-time software due to their advantages in flexibility and expressiveness. When realizing UML ports in object oriented languages, using objects for each port is one option. However, this approach causes runtime overhead and renders significant amount of additional generated code. To meet the performance constraints and decrease the costs of code reviews required in development of safety-critical real-time embedded software, more efficient approaches are required. In this article, we propose an approach, which introduces relatively less runtime overhead and results in more compact source code. A structural model defined with UML ports is transformed into a model that uses associations instead of objects to efficiently implement the UML port semantics with less lines of code. Achieved improvements and validation of the proposed approach is demonstrated by a case study; the design of an existing avionics software.

1 INTRODUCTION

Defining interactions of classes using association relations which expose the entire public interface of the classes to their clients makes it hard to observe the data flow in the model. Since ports function as an opening in the encapsulation of classes through which messages are sent either into or out of the class (Bjerkander and Kobryn, 2003), a model designed using ports is easier to understand, more flexible, easier to maintain and more suitable for communication of design decisions. In UML, a port is a point defined by a classifier for conducting interactions between the internals of the classifier and its environment. The contract based interaction provided by ports allows the classifiers to be defined independently from other classifiers (Selic, 2003).

On the other hand, software for safety-critical real-time embedded systems is becoming more and more complex (McDermid and Kelly, 2006). These systems typically require certification and runtime overhead of any design decision should be justified because of limited resources. Additionally, the size of the source code should be compact for easier code reviews. For instance, DO-178C (RTCA, 2011) defines several requirements for certification of airborne systems. In DO-178C, if a development tool (i.e. a code generator) is not "qualified", its outputs must be manually reviewed for correctness. Since development tool qualification is the most rigorous type of tool qualification in the scope of DO-178C, often, instead of qualifying the tools, their outputs are preferred to be manually reviewed. As a result, source code, which is more compact and less complex, is preferred because the effort needed for code reviews is expected to be lower.

Ports and connectors do not have direct correspondents in object oriented programming languages (Mraidha et al., 2013) and yet, UML does not put constraints on how ports are realized (France et al., 2006). The UML standard mentions ports as interaction points which provide "unique references" (OMG, 2015). According to this definition, realization of ports using objects seems adequate. However, this approach causes certain amount of runtime overhead for the final executable (Douglass, 2007). Ideally, ports should have zero overhead for transmitting messages for complex real-time systems (Selic, 1998). Another problem with this approach is, source code grows significantly because of the added objects and classes to realize the ports.

In this article, we propose a more efficient approach for mapping ports to object oriented languages. The proposed approach enables

Kocataş, A., Can, M. and Doğru, A.
Lightweight Realization of UML Ports for Safety-Critical Real-Time Embedded Software.
DOI: 10.5220/0005689602580265
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 258-265
ISBN: 978-989-758-168-7

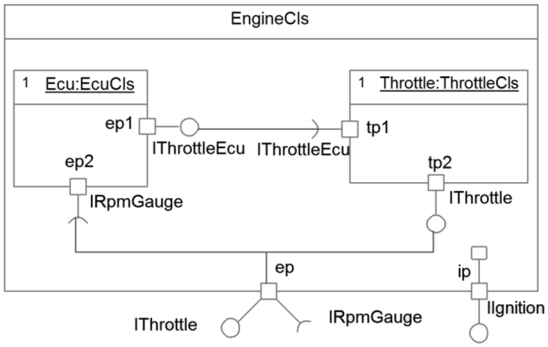

Figure 1: UML composite structures and port notation.

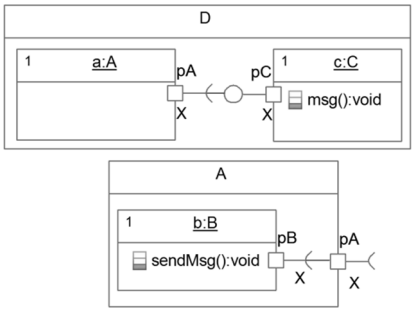

Figure 2: Example source model.

generation of relatively compact source code and introduces relatively less runtime overhead. A source model having been defined using ports and connectors is transformed into a target model. Code is then generated from the target model. The target model includes association relations and initialization operations, which implement the ports and connectors in the source model. The presented approach is evaluated using the design model for an avionics software.

2 UML PORTS OVERVIEW

Abstract syntax and concrete syntax for composite structures and ports is given in the "Composite Structures" section of UML 2.4.1 superstructure definition (OMG, 2015). Figure 1 presents an example composite structure diagram using ports. In the figure, there is a connector which connects the ports *ep1* and *tp1*. The connector semantically indicates that a message sent from the *tp1* port of the *Throttle* object should be sent to the *Ecu* object via its *ep1* port. When sending the message, source class (*Throttle*) specifies the port name, operation name and operation arguments. An example expression is presented in Statement 1 using C++:

```
GetPort(tp1).msg();          (1)
```

`GetPort` is used to obtain a reference to the destination of the message, and it should be translated to an appropriate statement by the used port realization approach. In this article, connectors are categorized as *cross connectors* and *relay connectors* for better communication of ideas. Cross connectors connect ports of the two objects (i.e. the connector between ports *ep1* and *tp1* in Figure 1), while relay connectors connect a port of an internal part with a port of the owner class (i.e. the connector between ports *tp2* and *ep* in Figure 1).

3 APPROACHES FOR REALIZATION OF PORTS

In order to demonstrate different realization approaches for ports, the model depicted in Figure 2 is used. In the model, a message is transferred from object *b* to *c* via the path, which is formed by ports *pB*, *pA* and *pC*. In addition to the example model transformations presented in the article, UML model for Figure 1 and the source code generated using all of the presented approaches are presented in a public repository[1]. The C++ language is chosen as the target language, but the approaches are also applicable to other object oriented languages.

3.1 Heavyweight Approach

Specific realization of ports as presented in this section is implemented by one of the existing modeling tools (IBM, 2015). According to this approach, each port is transformed to a class and its corresponding object. Furthermore, in order to differentiate messages from both directions, provided and required parts of port contracts are realized using *in* and *out* objects, which are instances of additionally generated classes. Using this approach, the example model provided in Figure 2 is transformed into the model shown in Figure 3. *PA, PB,* and *PC* are the classes generated for ports. Classes *Out* and *In* are generated for outbound and inbound direction of ports. After the transformation, the constructor for class *C* includes:

```
pC.getIn().setItsX(this);          (2)
```

Statement (2) is required to connect port *pC* of class *C* to object *c*. Furthermore, the constructor of class *A* is generated as:

[1]https://github.com/alperkocatas/UmlPortStudy

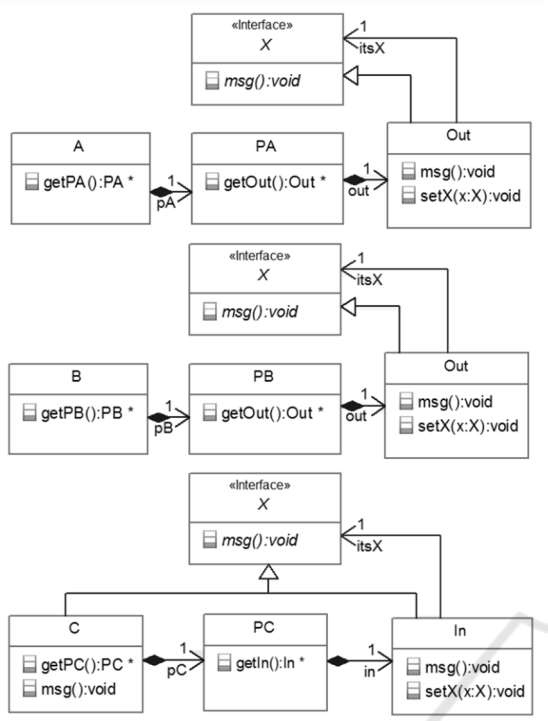

Figure 3: Realization of ports with heavyweight approach.

```
b.getPB().getOut().
    setX(getPA().getOut());          (3)
```

The method *setX* is a trivial accessor function for the generated associations *itsX*. Statement (3) is required to connect the relay port *pA* of class *A* to the port *pB* of object *b*. Finally, constructor for class *D* is generated as:

```
a.getPA().getOut().
    setX(c.getPC().getIn());         (4)
```

Statement (4) initializes the *itsX* association of *out* object of port *pA* with the reference of the *in* object of object *c*, so that messages sent from port *pA* will be handled by the *in* object of port *pC* of object *c*. `GetPort(PortName)` in Statement (1) is translated as `getPortName()->getOut()`, which returns a reference to the *out* object of the port. This reference is used to send messages from the port.

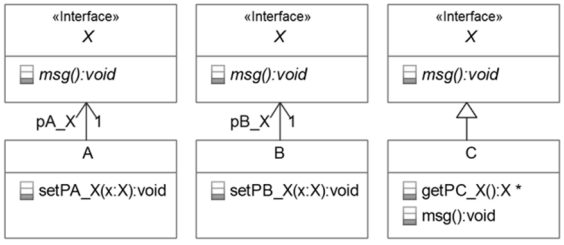

Figure 4: Lightweight realization of ports.

3.2 Lightweight Approach

In the lightweight realization, no objects are generated for ports. Instead, only associations with required interfaces and the operations to initialize them are generated. Relay connectors are particularly transformed into smart getter and setter methods. The smart getters and setters are used to connect the ports which are at the ends of a chain of relay connectors. At runtime, after the initialization code generated for cross connectors runs, each object enters a state in which, the final destinations of the messages that will be sent from its ports are determined. As a result, when sending messages, relay port chains spanning multiple ports are resolved in one step. Using this approach, the example model in Figure 2 is transformed to the model, shown in Figure 4. In the transformed model, associations *pA_X* and *pB_X* are used by objects *a* and *b* to send messages through their ports *pA* and *pB*. Since none of its ports requires interfaces, class *C* does not have such association. Constructor of class *D* is generated as:

```
a.setPA_X(c.getPC_X());              (5)
```

Statement (5) is generated for the cross connector in the source model. It initializes the *pA_X* association of object *a* with the value obtained from the *getPC_X()* operation of object *c*. Since *pC* is a behavioral port, generated operation *getPC_X()* returns *this* pointer of object *c*. The body of operation *setPA_X(X& val)*, which is a smart setter, is generated as:

```
pA_X = val;                          (6)
b.setPB_X(val);                      (7)
```

Port *pA* is a relay port, which forwards messages from *pB* to *pC*. Statement (6) first sets the *pA_X* association of object *a* to *val*, so that object *a* can also send messages using the port *pA*. Statement (7) then forwards the parameter *val* to object *b*, which will set association *pB_X* of object *b* to *val*. Since the value of *val* is passed as the *this* pointer of object *c*, Statements (6) and (7) effectively connect the port *pB* of object *b* with object *c*. As a result, after the expression in Statement (5) is executed, destination of messages going out from port *pB*, which is the object c, is determined. In the lightweight approach, the expression `GetPort(PortName)` in Statement (1) is translated as `PortName_InterfaceName`, which is the name of the association created for the port and required interface pair. To resolve the interface which is appended to the port name, an interface, which implements the operation being

called is searched in the required interfaces of the port. This search is performed at the time of port realization.

3.3 Disconnected Ports and Interfaces

If the source model contains disconnected ports, problems may occur at runtime. The following two cases correspond to disconnected ports: First, if a port is not connected to any other port via a connector, the port is considered as *disconnected*. Second, when there is a connector which connects two ports, if the ports at both ends of the connector do not have matching contracts, ports are considered as partially disconnected. In this paper, the second case is denoted with the term *disconnected interfaces*. When operations declared by the unmatched interfaces are called, messages cannot be forwarded because there is no provided interface at the opposite side of the connector. Figure 5 depicts an example for disconnected interfaces. The port *pA* requires interfaces *X* and *Y*, while the port *pB* provides interfaces *Y* and *Z*. Even if the ports are connected with a connector, since the port *pB* does not provide interface *X* in its contract, object *a* cannot send messages declared by interface *X* to object *b*.

In the lightweight approach, if the source model has disconnected ports or disconnected interfaces, messages which are being sent from the generated associations may cause null pointer exceptions. One of the following strategies can be employed for handling disconnected ports and interfaces:

1 - The lightweight approach can be used and disconnected ports and interfaces can be allowed in the model. Then, if there is a call from disconnected ports or interfaces, software may crash at runtime because of a null pointer exception. Alternatively, the model can be checked before realization of ports to ensure that no messages will be sent using disconnected ports or interfaces during execution.

2 - The lightweight approach can be used but, disconnected ports and interfaces are not allowed in the source model. The model can be checked before the realization of ports to ensure that there are no disconnected ports or interfaces.

3 - The lightweight approach can be modified so that all of the messages sent through disconnected ports and interfaces are handled at runtime, by ignoring the message or by throwing an exception.

When the first option is used without model checking, there is a possibility of software crashing at runtime due to a null pointer exception. However, if statement coverage for all of the operation calls

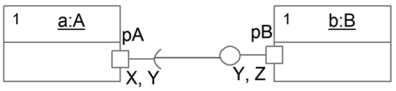

Figure 5: An example for disconnected ports/interfaces.

from ports is achieved while testing, it can be guaranteed that no such crash will occur. For example, DO-178C (RTCA, 2011) requires full statement coverage by test cases, beyond a certain safety-criticality level. Therefore, effort required for achieving full statement coverage is already included in the development costs. Alternatively, instead of achieving the statement coverage, the model checking mentioned in the first option can be attempted to be incorporated. However, such model checking is not trivial, since predicting the dynamic runtime behavior of software may not be feasible.

The second option is possible to implement. However, it can be argued that disconnected ports and interfaces are part of the flexibility offered by ports. For example, in Figure 5, although object *a* cannot send messages declared by interface *X*, it can be allowed to send messages declared by interface *Y*. Thus, if the operations declared by the interface *X* are not crucial for the expected behavior of object *a*, the model in Figure 5, which will is invalid according to the second option, can be assumed as a valid one.

The third option can retain the flexibility of ports, while providing graceful runtime error handling. Indeed, this option is supported by the presented heavyweight approach. When a separate object is employed for each port, they can check whether the destination of a message is null at runtime. As a result, one of the three options can be used. Because it allows more flexibility during modeling, the third option was selected for implementation. Next section presents how to apply

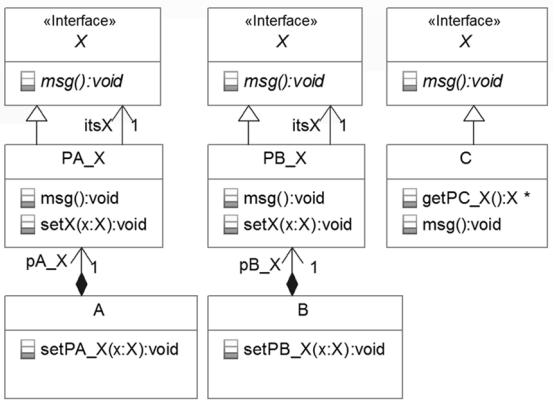

Figure 6: Lightweight realization of ports with checks for disconnected ports.

the third option to extend the lightweight approach.

3.4 Extending the Lightweight Approach for Disconnected Ports

In this approach, a port with a required interface is implemented using a separate object, which can check if its generated association is null at runtime. Ports having only provided interfaces in their contracts are still implemented as they would be in the lightweight approach, without any objects. Generation of smart getters and setters for the relay connectors is also the same as in the lightweight approach. Using the approach, example model presented in Figure 2 is transformed to the model shown in Figure 6. Associations *pA_X* and *pB_X* in the lightweight approach are generated as objects. The objects have associations with interface *X*, called *itsX*. The port connection initialization statements generated in the constructors are identical with the previous approach. However, now the *msg()* operation, which is implemented in the classes *PA_X* and *PA_B* objects checks if the *itsX* association is *null* at runtime. In this extended version of the lightweight approach, the expression `GetPort (PortName)` in Statement (1) is translated as `getPortName_InterfaceName()->getOut()`.

4 EVALUATION

4.1 Performance Analysis

One of the important performance drawbacks of ports originate from messages which are redirected through multiple ports along the path of the messages. The lightweight approach ensures that at runtime, the objects hold a reference to the final destinations of the messages. In this approach, messages are directed to their final destinations in one step. In the heavyweight approach, messages cannot reach their destinations in one step, but they are redirected multiple times between ports along the path of the message. When the extension for disconnected port checking is added to the lightweight approach, performance is still expected to be better than the heavyweight approach because only one level of redirection is added.

Avoiding unnecessary message redirections also may yield faster performance due to the decrease in the number of instruction pipeline operations. Furthermore, the computation required for the creation of objects during initialization is another potential drawback for the heavyweight approach. The lightweight approach and its extension are expected to perform better since the number of created objects is zero or at least, fewer.

4.2 Code Size Analysis

Lightweight approach is expected to deliver the most compact code among the presented approaches because, no objects and classes are generated for ports. When the checks for disconnected ports and interfaces are added, the code size should not grow as much as it would in the heavyweight approach. This is because objects are only generated for the outbound directions of ports. In the heavyweight approach, resulting code size is considerably larger, due to the implementation of the operations declared by the provided and required interfaces in every class created for ports. Smaller code size is also expected to improve cache utilization, thus improving the runtime performance.

4.3 Case Study Design

In order to compare the results from both approaches, a previously released version of an avionics software was used. The software was developed by a team of around fifty software engineers within a five year schedule, and is still in progress. The software coordinates more than thirty avionics devices and provides the pilot with crucial flight information. IBM Rhapsody (IBM, 2015) is being used as the development tool. DO-178C (RTCA, 2011) is complied. Because the code generator of IBM Rhapsody is not a “qualified” development tool in the scope of DO-178C, generated code is reviewed manually.

Software components in the case study run on an ARINC 653 (Aeronautical Radio Inc., 2003) compliant real-time operating system. Software processes run in time and space isolated *partitions*. Partitions are scheduled in a fixed, cyclic basis. Partitions perform their initialization tasks and then start executing a periodic running task. When the allocated execution duration finishes for the scheduled partition, the scheduler switches to the next partition in the schedule.

For comparison purposes, code was generated using different approaches. Several metrics were collected during code generation and runtime. AVGRT metric indicates the average time in milliseconds, required for a partition to finish its periodic execution. The LSLOC metric indicates the logical source lines of code measured using the

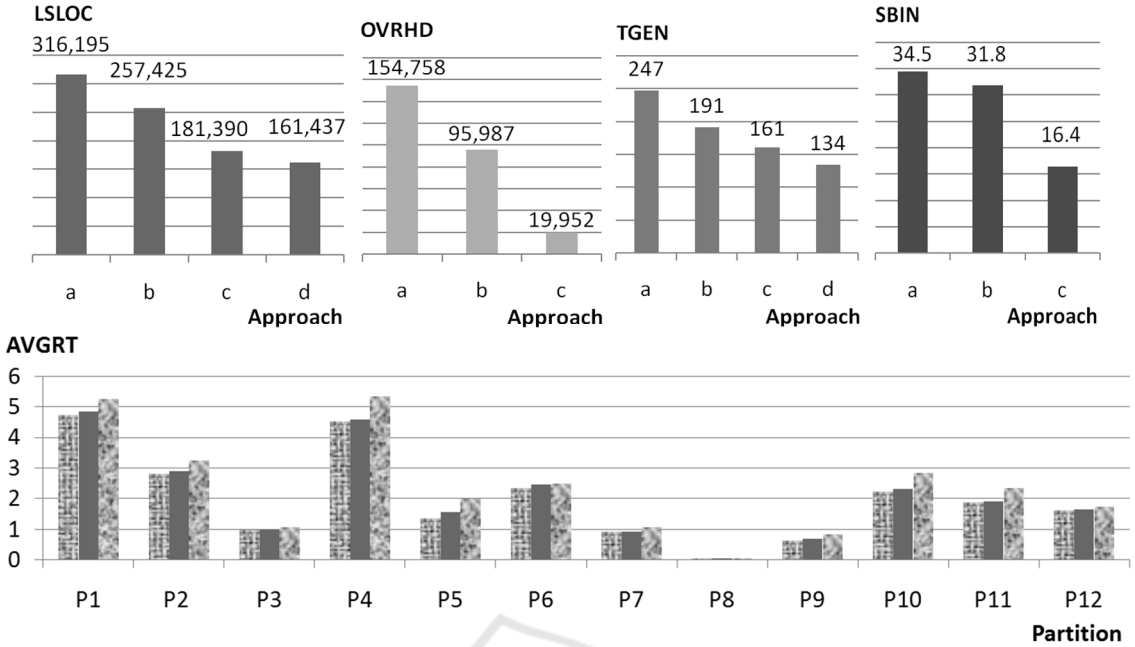

Figure 7: LSLOC, OVRHD, TGEN, SBIN and AVGRT metrics - Approaches: (a) Heavyweight approach, (b) Lightweight approach with checks for disconnected ports, (c) Lightweight approach, (d) None (ports are not realized). AVGTR metric - Left bar: Lightweight approach., Middle bar: Lightweight approach with checks., Right bar: Heavyweight approach.

Unified Code Count tool (Nguyen et al., 2007).

LSLOC is not affected by style and formatting decisions. OVRHD metric is defined as the difference between LSLOC measurements for the source code with and without port realization. Thus, the OVRHD metric indicates how much LSLOC increase is induced by port realization. The TGEN metric indicates the time in seconds, required to realize the ports in the source model, plus the time required to generate source code from the target model. SBIN metric indicates the size of the final executable binaries in megabytes.

4.4 Results and Discussion

Figure 7 shows charts which correspond to the measurements for given metrics according to different approaches. Using the lightweight approach, LSLOC was dramatically reduced from 316,195 to 181,390. The 42% decrease in the size of source code may provide significant cost saving during code reviews. Likewise, OVRHD metric is the lowest for the lightweight approach. When the checking for disconnected ports is incorporated, OVRHD is still significantly less than the OVRHD measured for the heavyweight approach.

According to the TGEN measurements, the fastest functional code generation was available with the lightweight approach, which is followed by the lightweight approach with checks. SBIN metric measurements showed that resulting binaries were most compact with the lightweight approach.

After generation and compilation of source code, binaries were executed. Generated code ran successfully. Verification of the build was demonstrated by running a subset of the test cases used in the formal software release. With the lightweight approach, no null pointer exceptions due to disconnected ports were observed at runtime. This was not surprising since the previous release of this software was already tested and full statement coverage was achieved for statements used to send messages from ports.

During collection of the AVGRT metrics, identical set of functions were activated in order to generate identical load on the system. According to the results, up to 32% improvement was observed for the AVGRT metrics. When the runtime improvements for each partition were averaged, 15.7% over-all performance improvement was observed over the heavyweight approach by the lightweight approach. A similar calculation revealed an average of 10.9% performance improvement by the lightweight approach with disconnected port checking over the heavyweight approach. The variation of improvement percentages for each partition is due to the different levels of port usage and composite structure hierarchies. The results

showed that using the lightweight approach and its extension not only yields more compact code, but also yields faster execution when compared to the heavyweight approach.

5 RELATED WORK

The concept of ports is also available in other modeling languages. UML-RT (Selic, 1998) and MARTE (OMG, 2011) are two of the UML extensions for modeling safety-critical embedded real-time software. In UML-RT, a *protocol* defines which kind of messages can be received and sent from a port. In UML, the *provided* and *required* properties of ports capture the information captured by the protocol concept. On the other hand, MARTE categorizes ports as client/server ports and flow ports. Client/server ports are a syntactic sugar over UML ports, enabling a more convenient way to define the port contracts. The flow ports are used to model the data flow between structured components. Flow properties and flow specifications are used to define the messages which can flow through the ports. If flow specifications and properties are mapped to interfaces, flow ports can be realized using the approaches presented in this article, or else they may need different realization approaches. Based on the purpose of port representation and realization, such alternative languages do not offer additional advantages. Consequently, we have exploited UML 2.0 due to its wider usage and our access on vast project material.

UML ports and composite structures are mentioned in previous studies, which use ports to model embedded systems. The ports are mapped to target languages such as SystemC (Andersson, 2008), (Xi et al., 2005), VHDL (Vidal et al., 2009) and Simulink (Brisolara et al., 2008). In these studies, one-to-one mapping between UML ports and the target languages could be performed since the target languages provided constructs, which correspond to UML ports.

For mapping ports into object oriented programming languages, there are studies which suggest mapping ports using objects (Willersrud, 2006). The heavyweight approach, which is presented in this paper, is employed by IBM Rhapsody (IBM, 2015). To cope with the performance degradation, IBM Rhapsody offers an optimization, which is performed at runtime to find the final destinations of ports (Douglass, 2007). The optimization runs during the initialization of software and uses algorithms to traverse the relay connector chains to find the ultimate targets of the messages. However, the additional computation during initialization and the use of special data structures make the code even more complex and harder to review.

Possibility of a lightweight realization of ports was mentioned previously (Bock, 2004). It was argued that ports need not to be realized as objects, but they can also be realized in a lightweight fashion, with no port objects created at runtime. However, the study did not present a specific method for the mentioned lightweight realization of ports. Another approach for realization of ports (Mraidha et al., 2013) is very similar to the lightweight approach presented in this paper. However, the approach creates the getter methods using only the name of the ports, without utilizing the interface names. This naming scheme cannot cope with cases where a source port requires more than one interface in its contract and it is connected to more than one destination ports, each destination port providing one of the different interfaces required by the source port. Moreover, the validity of the approach was not demonstrated by a case study or other means.

6 CONCLUSIONS

This article proposed a lightweight approach for mapping UML ports to object oriented programming language constructs. The article first presented a widely used method for realization of UML ports, which is prone to performance and code size problems. Afterwards, the lightweight approach, which enables the use of ports without sacrificing runtime performance and source code size, was presented. Additionally, the problems which may be caused by disconnected ports are discussed and an extension to the lightweight approach, which can be used for handling disconnected ports in the source model was presented.

Presented approaches were compared using metrics collected from a real-life case study. Metrics used for comparison are logical source lines of code, average runtime performance, model transformation duration and binary size. The case study showed that the proposed lightweight approach results in more efficient and more compact code. Additionally, the time required for realization of ports and the size of executable binaries produced after compilation were also lower with the proposed approach. Better performance may yield more headroom for meeting hard real-time requirements, while smaller and less

complex code may enable relatively easier and accurate code reviews, potentially improving safety.

REFERENCES

OMG, 2015.*Unified Modeling Language, Superstructure*, Version 2.4.1, http://www.omg.org/spec/UML/2.4.1/, last accessed on Sep 16th, 2015.

J. Vidal, F.,Lamotte, G.,Gogniat, P. Soulard, Diguet, J. P., 2009. A co-design approach for embedded system modeling and code generation with UML and MARTE, *Design, Automation & Test in Europe Conference & Exhibition*.

Andersson, P. 2008. UML and SystemC – A Comparison and Mapping Rules for Automatic Code Generation. In:Villar, E. Ed. *Embedded Systems Specification and Design Languages*. Springer Netherlands.

Xi, C., Hua, L., J., Zucheng, Z., YaoHui, S. 2005.Modeling SystemC design in UML and automatic code generation, *Proceedings of the 2005 Asia and South Pacific Design Automation Conference,* pp 932-935.

McDermid, J., Kelly, T. 2006. Software in Safety Critical Systems: Achievement and Prediction, *Journal of The British Nuclear Energy Society, Vol. 02, pp. 140-146.*

France, R., B., Ghosh, S., Dinh-Trong, T., Solberg, A. 2006. Model-driven development using UML 2.0: promises and pitfalls, *Computer,* Vol. 39, pp 59-66.

Mraidha, C., Radermacher, A., Gerard, S. 2013. Model-Based Deployment and Code Generation. In: Hugues, J., Canals, A., Dohet, A., Kordon., F. ed. *Embedded Systems: Analysis and Modeling with SysML, UML and AADL*, Wiley, 2013.

Willersrud, A. 2006.User-defined Code generation from UML 2.0. *(M.Sc. Thesis).* Permanent link: http://urn.nb.no/URN:NBN:no-12371.

IBM. 2015. Rational Rhapsody Developer, 8.1.1, http://www-03.ibm.com/software/products/en/ratirhap, last accessed on May 8th, 2015.

Douglass, B., P. 2007. Systems Architecture, In: *Real Time UML Workshop for Embedded Systems*, ISBN: 978-0-7506-7906-0. Newnes, 2007, pp:90-92.

Selic, B. 1998.Using UML for Modeling Complex Real-Time Systems, *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*. Springer-Verlag, 1998.pp. 250--260.

Bock, C. 2004.UML 2 Composition Model. *Journal of Object Technology*, Vol. 3, No. 10.

Bjerkander, M., Kobryn, C. 2003. Architecting Systems with UML 2.0, *Software*, IEEE Volume:20 , Issue: 4, pp. 57 - 61.

RTCA, 2011.DO-178C Software Considerations in Airborne Systems and Equipment Certification.

Selic, B. 2003.Architectural Patterns for Real-Time Systems, In: Lavagno, L., Martin, G., Selic., B. ed. *UML For Real.* Kluwer Academic Publishers, Norwell, MA, 2003, USA 171-188.

Aeronautical Radio Inc. 2003.653-1 Avionics Application Software Standard Interface.

Nguyen, V., Deeds-Rubin, S. Tan, T., Boehm, B. 2007.A SLOC Counting Standard, *COCOMO II Forum*.

Brisolara, L., B., Oliveira, M., F., S., Redin, R., Lamb, L., C., Wagner, F. 2008. Using UML as Front-end for Heterogeneous Software Code Generation Strategies, *Design, Automation and Test in Europe*, pp.504,509.

OMG, 2011.*UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems*, Version 1.1 http://www.omg.org/spec/MARTE/1.1/, last accessed on Sep 16th, 2015.