Model Driven Engineering of Communication Protocol Artifact with Design Pattern Usage in Distributed and Real-Time Embedded Systems: An Industrial Experience

Deniz Akdur*‡, Ebru Özpolat**, Tuna Başıbüyük**

* ASELSAN Inc., Ankara, Turkey

* Department of Information Systems, Informatics Institute, METU, Ankara, Turkey

** ASELSAN Inc., Ankara, Turkey

(denizakdur@aselsan.com.tr, eozpolat@aselsan.com.tr, tturk@aselsan.com.tr)

‡ Corresponding Author; Deniz Akdur, ASELSAN, Gölbaşı, Ankara, Turkey, Tel: +90 312 592 6000/82183,

Fax: +90 312 592 6006, denizakdur@aselsan.com.tr

Received: 05.08.2017 Accepted:21.09.2017

Abstract- Distributed and real-time embedded systems, in which collections of independent computers interoperate, appear to users as a single coherent system by creating "systems of systems". The scale and complexity of such systems makes it infeasible to deploy them in standalone configuration, which highlights the necessity of more systematically designed and implemented communication protocol assets. These assets should be not only close-to error proneness with abstraction, but also reusable to achieve maximum efficiency and effectiveness during software development life cycle. To address these challenges in a specific industrial context, we designed and implemented reusable artifact for communication protocols via model driven engineering with the help of design pattern usage. This artifact is currently in use by many teams in the company as we report its solution approach and its impacts in this paper.

Keywords Model driven engineering, design pattern, distributed, real-time, embedded.

1. Introduction

Distributed and Real-Time Embedded (DRE) systems, in which collections of independent computers interoperate, appear to users as a single coherent system [1]. They are combined with other embedded systems to create “systems of systems” [2]. Since the scale and complexity of DRE systems makes it infeasible to deploy them in disconnected, standalone configurations [3], communication is at the heart of all distributed systems. An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalized in what are called protocols [1]. To allow a group of computers to communicate over a network, they must all agree on the protocols to be used.

The latest modern DRE systems have numerous components, in which there are various kinds of communication protocols. ASELSAN [4], one of Turkey's leading defense companies, develops hybrid systems by combining various radars and electronic warfare systems in a single product. These products have a large number of both

internal and external interfaces with different components like transmitters, receivers, power supplies, antennas, display units via different communication protocols such as serial channels (i.e., RS232, RS422) VxWorks Message Queue, TCP, UDP, Peripheral Component Interconnect Express (PCIe) [5].

When the number of projects was less and there was no need to care about reusability and portability, projects' communication protocol assets were developed independently. In that case, every project had different assets for the same purpose (i.e. every project had its own TCP or Serial Channel implementation). Due to the major increase in complexity and size of the products, it was very crucial to have reusable communication protocol assets, to achieve maximum efficiency and effectiveness [6] during Software Development Life Cycle (SDLC). This was not a crucial need for the implementation, but also maintenance. For the implementation phase, developers started to complain about unnecessary waste of time on reinventing the wheel for a specific communication protocol. Moreover, in the maintenance phase, there was no chance for a new coming engineer to easily and quickly understand the different communication protocol implementations although he/she took part in such a similar implementation in another project. In summary, developers wanted to spend most of their time and effort on the actual system scenarios ("business logic") instead of standard communication protocol implementation. If there would be an asset not only a reusable, but also close-to error proneness with abstraction [7], we would also achieve maintainability and portability.

With all of these motivations, the importance of more systematic software engineering practices were highlighted and to address these challenges, an R&D project has been started to design and implement a reusable artifact for communication protocols, in which Model Driven Engineering (MDE) is used. In order to get rid of accidental complexities during implementation [8], our Model Driven Development (MDD) approach, which is a subset of MDE, is enriched with design pattern usage by making it easier to reuse successful designs and architectures, which also support extensibility [9]. On one hand, the user of this artifact can benefit from it as an pre-built library without knowing its details via just implementing its given interfaces [5]. On the other hand, the user can integrate the output of this artifact as a reference project in IBM Rational Rhapsody Developer for C++ tool [10], which is also used as MDE tool in the company for automatic code generation. In that situation, the user can understand the underlying mechanisms by seeing and analyzing the necessary UML models within the artifact.

Software documentation is very important not only in the process of implementation, but also in test and maintenance [11] since careful documentation can save an organization's time, effort and money [12]. The emergence of wide spectrum of embedded systems, and the increasing use of software for implementing the functionality, has led to increasing demands for more sophisticated embedded software maintenance, which highlights the importance of good documentation [13]. This problem can be solved with the help of MDE [11] and we achieved this via our MDE tool, in which we implemented our MDD approach. While implementing our asset, we carefully comment our design, asset usage and key points; then the generated code and documentation is always synchronized with the help of MDE.

This communication protocol artifact is currently in use by many teams in the company as we report in this paper.

The remainder of this paper is organized as follows. Section 2 presents the literature review. In Section 3, the solution is presented. Section 4 examines the evaluation of our solution in the industrial context, in which the applicability and usefulness of the approach are shown. Finally, Section 5 presents conclusion.

2. Literature Review

As software projects grow in scale and scope, the extensibility of existing software and reusing existing code are gaining more importance [6, 9]. There are several industrial experiences on the benefits of software reuse [14-16]. "Reuse" lowers development costs by reducing development time, increases reliability and also reduces process risks [17]. In addition to this, it is generally agreed that the most common realistic way to manage the software complexity is developing it using appropriate methods of abstraction [18]. Modeling is seen as a way to better handle this growing complexity of software development by helping engineers to work at higher levels of abstraction and facilitates communication [19].

Nowadays, the state-of-the-art in software abstraction is MDE [20], which can be seen as the systematic use of models as primary artifacts during a software engineering process. MDD, as a subarea of MDE, has recently become a hot topic in both industry and academia. There are several books, e.g., [21-24], many research articles, e.g. [25, 26], many reports, e.g. [27, 28] in the development and application of model-driven technology for DRE systems.

On the other hand, expressing proven techniques as design patterns help the designer choose among design alternatives that make a system extensible and avoid alternatives that compromise reusability [29]. A design pattern is a general reusable solution to a commonly occurring problem in software design [30]. Although software patterns do not address extensibility explicitly, almost every pattern that supports changeability also supports reusability and extensibility [9]. In fact, a design pattern is a way of reusing abstract knowledge about a problem and its solution [31], which can be seen as successful family of proven solution to a recurring problem that arises within a certain context [17]. There are also several books and many research articles [32-36] on design pattern usage in embedded systems.

Although there are some communication protocol studies, generally, they do not make use of object-oriented frameworks and they use the facilities offered by the Socket API, which has severe limitations, and is considered a complex, non-portable and error-prone [37]. On the other hand, despite its necessity, there are few studies to exploit the benefits of MDE on communication protocol implementation, but either to simplify or manage the design of specific system architecture (i.e. network services for the Internet, which is based on a specific Client-Server architecture [38] or manage

communications with and between Resource-Constrained Systems within a heterogeneous Wireless Sensor Network (WSN) [39]). Apart from these, there are also some attempts to create a consistent set of interfaces and APIs for communicating embedded systems as a communication layer (i.e. the Multicore Association's MCAPI/MRAPI [40]). In our MDE approach, which is enriched by design pattern usage, we achieve not only portability, maintainability and non-error proneness but extendibility by supporting not only a specific architecture (i.e. Client-Server) but also some others (i.e. serial channels or PCIe) according to our needs.

3. Solution

3.1. Selection of the solution approach

Early in the project, after we identified the challenges and needs, the first immediate step was to list the candidate solution approaches from the software engineering domain applicable to the problem. After conducting a literature review to see if approaches or tools applicable to our context have been proposed before, we saw that if MDD is enriched with software design patterns, which help designers build on the collective experience of skilled software engineers by capturing existing, well-proven experience, the impact of such a systematic software engineering approach would be maximized in SDLC. In other words, reusing the code and reusing experience complete each other in MDD with design pattern usage. Thus, we decided our solution approach as MDD of communication protocol artifact with design pattern usage. We discuss this approach's design and development aspects next.

3.2. Design and development aspects

In our problem domain, the communication-related components should:

- Establish the connection according to protocol type and corresponding settings
- Marshall data
- Send data
- Receive data
- Unmarshall data
- Close the connection

As an architectural design decision, the layered architecture [41] is used in our embedded software projects. These layers are mainly L1 (Communication Protocol), L2 (Communication Middleware) and L3 (Functionality), which deals with "business logic". The artifact presented in this paper corresponds to L1, which handles with all the above operations except "Marshall" and "Unmarshall" operations, which are the responsibilities of L2.

3.2.1Modeling the real-time scenario via MDD

Due to real-time requirements during receiving data operations, the user of this artifact must be informed as soon as a connection is established or any data arrives. In order to realize this, a four-state-statechart is modeled in our solution domain as in Fig. 1. In the Initial state, nothing is done until a connection request. With this connection request, EstablishConnection state becomes active, in which necessary settings are arranged in order to communicate. A success scenario leads to WaitMessage state, where any incoming data is waited. A successive arrival of this incoming data is informed to the user of the artifact. Then, by reentering the same state, new messages are waited. During the EstablishConnection and WaitMessage states, any unsuccessful operation causes transition to Error state, in which connection is retried until an establishment or a closure request.

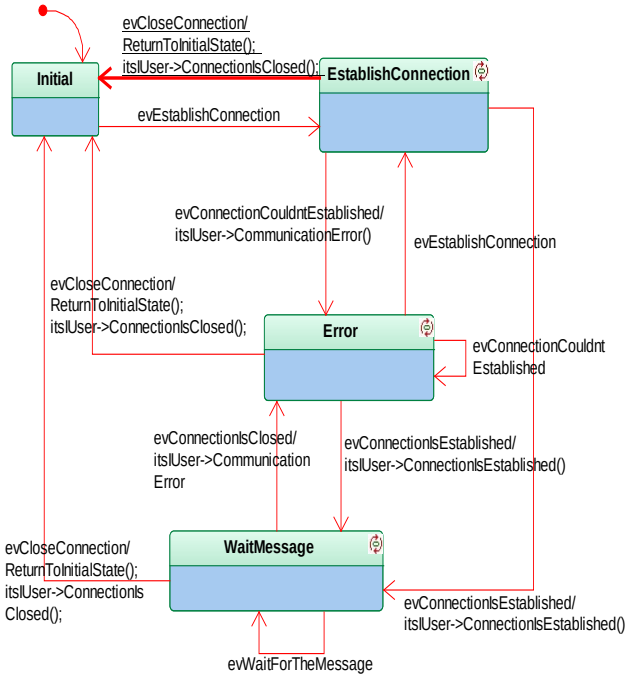

Fig. 1. The statechart used in MDD in our solution domain

The functions called in states or transitions between states of the statechart shown in Fig.1 are implemented in GeneralCommunication class, which introduces one more level of inheritance between ICommunication interface and specific protocol implementations as depicted in Fig. 2. The responsibility of GeneralCommunication class is to hold both the common functions, which are independent of the underlying protocol like "ManageError" and the functions, which are inherited in the underlying protocol and overridden accoding to the underlying protocol API like ReturnToInitialState(), SendMessage(), CloseConnection(), EstablishConnection() and ReceiveData(). ICommunication interface represents the interface between the user which stands in L2 in our architectural design Furthermore, the user in L2 is abstracted via IUser interface. As in our application domain, in defense domain projects, to provide the real-timeliness' of communication between software modules is very critical. Therefore, each instance of GeneralCommunication including the statechart figured out on Fig. 1. is coupled with a single active task in the operating system (OS). This statechart can be seen as the backbone of

the artifact, since it presents a common model for all communication protocol types and is inherited by every sibling of GeneralCommunication. With the help of this statechart abstraction, MDD manages the complexity in the communication protocol implementations without error-prone and accidental complexities by automating our artifact generation through productivity and interoperability [42].

3.2.2 Building the interfaces with bridge design pattern

When an abstraction has several possible implementations, accommodating them by using inheritance (i.e.ComponentAWithSerialChannel, ComponentAWithTCP, ComponentBWithSerialChannel…) might be an alternative solution. However, this approach isn't always flexible enough whenever the number of protocol types increases [29]. In our solution, two varying concepts that can be encapsulated are communication protocols, which are referred as implementation and the inter-dependent modules' specific interface requirements, which are referred as abstraction. In order to avoid a permanent binding between an abstraction and its implementation, we intended to hide the implementation details of the protocols by implementing the same interface via ICommunication class and hide the interface specific details by using IUser interface as shown in Fig.2. By this way, when GeneralCommunication is selected or switched at run-time, both its abstractions (i.e. IUser) and implementations (i.e. SerialChannel, UDP, TCP, etc.) are extensible by subclassing. In this case, the Bridge Pattern [29] helps to combine the different abstractions and implementations by extending them independently.

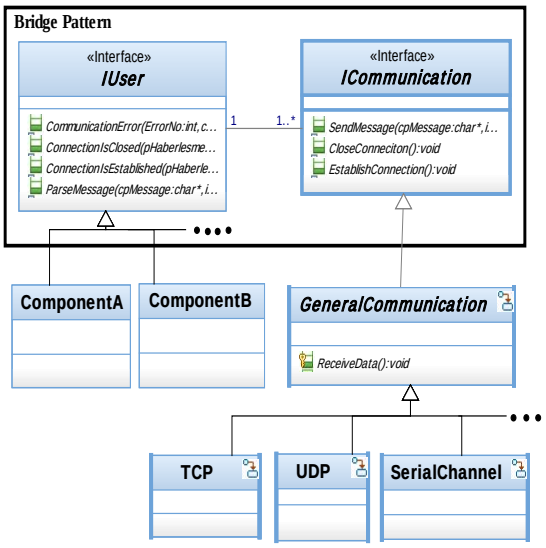

Fig. 2. The Bridge Pattern usage in our solution domain

New communication protocol types can be easily added by extending GeneralCommunication. By this way, we provide improved extensibility with the help of this pattern [29]. Moreover, the user, who implements IUser, knows neither the implementation details nor the type of communication protocol.

Siblings of IUser are independent from the underlying communication protocols and also provide the user standardized functions to implement project-specific behavior. These standardized functions are functions related to connection status and the function regarding parsing the received message. The implementation of these interface functions can vary depending on the inter-dependent modules' requirement; whereas siblings of ICommunication are already standardized (i.e. TCP Application Programming Interface (API)) and already implemented in our artifact, which makes them easier to be a reusable artifact. By this way, our artifact consists of IUser, ICommunication and siblings of ICommunication. From the user point of view, implementing IUser is sufficient to get benefit from this artifact.

3.2.3 Implementing protocols with adapter design pattern

In order to add a new communication protocol type implementation, SendMessage(), CloseConnection(), and EstablishConnection() interfaces, as shown in Fig.2, should be implemented according to the new protocol type API. Moreover, ReceiveData() function in GeneralCommunication should be overridden. For some protocols a level of inheritance is introduced in order to handle different usages.

As shown in Fig.3, TCP Server and Client usage are handled in different classes, similarly UDP Multicast and UDP Client. At that point, adapting the protocol APIs to ICommunication Interface is a challenging problem.

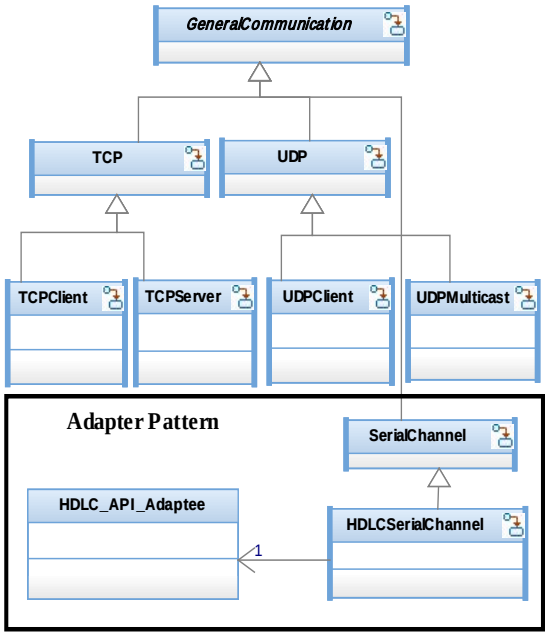

Fig. 3. The Adapter Pattern usage in our solution domain

"Establishing Connection", "Receiving Data" or "Sending Data" operations are straightforward and easy to implement in the well-known communication protocol types like TCP, UDP by being as a child of ICommunication. However, some memory based protocols like PCIe [43] or COTS PCI Mezzanine Card (PMC) modules might have different APIs, which are hard to encapsulate with ICommunication interface. High-Level Data Link Control (HDLC) is one such a challenging API and in order to overcome this challenge, we came up with Adapter Pattern with object composition [29] via HDLC_API_Adaptee as shown in Fig.3.

4. Evaluation in the industrial context

4.1. Usage

Due to confidentiality reasons, we are unable to report the application of our artifact on a real sub-system of the case-study projects. Instead, to demonstrate the applicability and usefulness of our approach, we report next its evaluation on a representative example.

In Fig.4, the representative example of object diagram for a standard user class is presented. In that situation, this user class (i.e. ComponentA, ComponentB) inherits from IUser and overrides CommunicationError(), ConnectionIsEstablished(), ConnectionIsClosed() and ParseMessage() functions for TCPClient and SerialChannel.

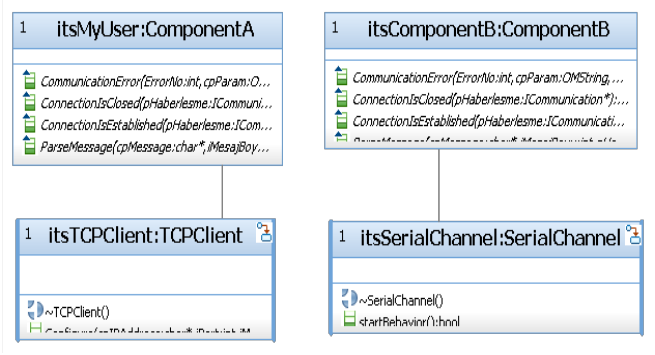

Fig. 4. The representative example of Object Diagram

In order to clarify the scenario, a communication diagram for a realistic scenario is given in Fig 5.

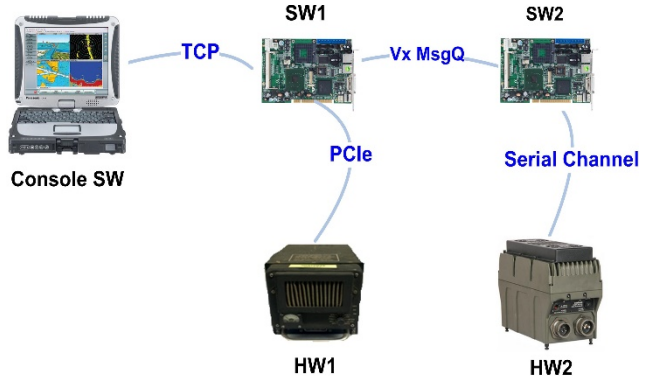

Fig. 5. Communication diagram for a realistic scenario

According to that scenario, as a display unit, Console SW sends a message to SW1 via TCP. After getting this message, SW1 sends a message (either changing this received data by applying some business logic or directly bypassing it) to HW1 via PCIe and SW2 via Message Queue. Then SW2, finally sends its message to HW2 via serial channel.

In order to understand the details of the realistic scenario above, a representative sequence diagram of our artifact usage is given just for one communication protocol between two units (either SW or HW) in Fig. 6. In this scenario, after creation of IUser and GeneralCommunication instances, necessary configurations depending on communication protocol type are made (In the scenario mapping, itsMyUser:IUser might be itsConsoleSW:IUser, itsSW1:IUser, or similar objects and comm1:GeneralCommunication might be itsTCP:ICommunication, itsPCIe:ICommunication, or similar objects). Then, after connection establishment, sending/receiving operations are occurred.

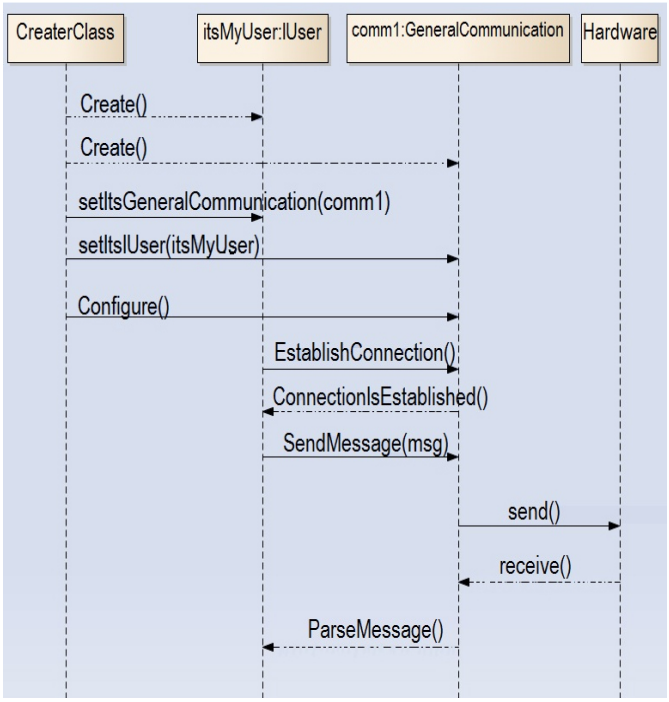

Fig. 6. The representative example usage of our artifact

4.2. Impacts, challenges and lessons learned

Our artifact had been designed and developed in ~490 person-hours and released to customers after its acceptance test process. On the other hand, its functionalities and capabilities have been still enriched with the feedback from users of this artifact by realizing their change requests (CRs) although it is in maintenance phase. As of March 2017, our artifact is used in 49 projects within 108 software configuration items (SCI) as shown in Table 1.

Table 1. Total subscription of this artifact

	Year							
	2010	**2011**	**2012**	**2013**	**2014**	**2015**	**2016**	**2017**
# of Projects	6	12	22	26	30	35	41	49
# of Users (SCI)	14	31	47	55	64	81	93	108

It is agreed that advanced middleware technologies [44, 45] by itself will not deliver the capabilities envisioned for next-generation distributed systems; therefore MDD, as a part of MDE, is needed not only to assist system developers in understanding their designs and but also to reduce the costs associated with trial and error by enriching interoperability [46]. The impact of MDD is very clear that the development time is very rapid when compared to ordinary scenario-based development. By this way, reusable artifact makes it easier to model and implement new CRs. Furthermore, verifying & validating a new communication protocol within the artifact is faster since all the implementations are derived from the same model.

Moreover, from the point of the user of our artifact, artifact adaptation to the project (i.e. linking the library or referencing the project with necessary configuration described in previous section) takes less than 1 person-hour without any extra training. It is obvious that the impact of this reusable artifact make the project gain a considerable amount of time on the communication protocol type related implementations.

In Table 2, data taken from the company's change management tool are presented.

Table 2. Data taken from change management tool

	Year						
	2010	**2011**	**2012**	**2013**	**2014**	**2015**	**2016**
# Reported Bugs	6	3	2	1	0	1	0
Time spent for bug fixing and verification (person-hour)	26.53	34.75	8.58	13.00	0.00	5.00	0
# Change Requests	2	14	1	2	1	0	0
Time spent for CR implementation and verification (person-hour)	21.00	246.5	6.17	61.05	10.00	0.00	0
Total time spent for bug fixing and CRs (person-hour)	47.53	281.3	14.75	74.05	10.00	5.00	0

By analyzing Table 1 and Table 2 data, it is seen that:

- The number of users of this artifact increases, whereas the number of reported bugs is decreased in time. Since we keep our artifact alive and up-to-date by dealing with reported bugs and CRs, the artifact has become more robust.

- In 2011, the number of users of this artifact becomes more than double. As a result, CRs make the peak, some of which causes major changes to our artifact (i.e. request for supporting both kernel mode and user space mode). For example, until that time, regarding VxWorks users, our artifact was running just in Kernel Mode. However, among these new users in 2011, some of them requested Real-Time Process (RTP) Mode from our artifact. This request made the development team spend lots of time while investigating RTP Mode usage. Therefore, such CRs, which are not related with our design approach, affected total time spent for CRs and this result in its peak value.

- In 2013, there was a considerable amount of time spent for total development and verification of this artifact although the number of reported bugs and CRs are not too much. In fact, the reason of this increase addresses the major challenges in embedded software development. Moving some projects to new embedded processors with new cores (i.e. to Intel processors with Pentium cores from PowerPC cores) required our artifact to support new functionalities and to extend our implementation without affecting the available capabilities besides changing our development framework.

  - Supporting both little-endian & big-endian architectures [47] and also both PowerPC and Pentium cores.

  - Upgrading embedded OS version (i.e., VxWorks), since the BSP (Board Support Package) of the processor supports only one version of the OS.

  - Serving different users for different OSs caused changes on the protocol API function and their usage. (i.e., in one VxWorks version (6.4) TCP keep parameters can be adjusted for whole sockets as a single adjustment; whereas in another version (6.8), TCP keep parameters can be adjusted for each socket separately).

  - Supporting/Upgrading all development framework since this is not backward compatible (i.e., all users are also IBM Rational Rhapsody [10] users).

Although there were above challenges, we coped all of them with the help of our MDD approach and benefited from design pattern usage when dealing with new CRs.

- In 2016, there was no reported bugs and CR on this artifact, which shows its robustness and maturity.

As a layered architecture, L1 and most part of L2 are automatically generated via MDD [5], whereas L3, which includes scenarios, is implemented manually. This shows that major part of our software is benefits model-driven approaches and close-to error prone characteristics of software complexities. Furthermore, using these reusable artifacts provides an already verified software module.

Table 1 and Table 2 data are visualized in Fig 7 graph.

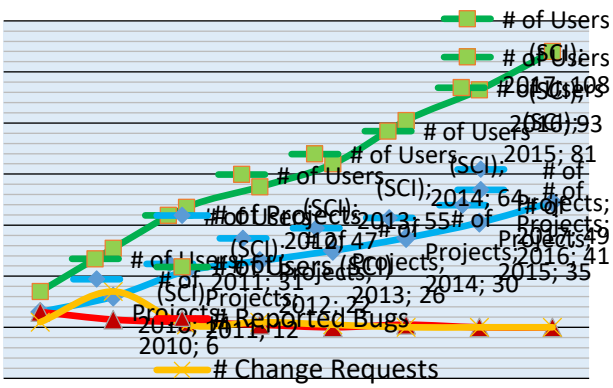

Fig. 7. Visualization of data from change management tool

5. Conclusion

The presented artifact for communication protocols based on MDE with design patterns is close-to error proneness with abstraction, but also reusable to achieve maximum efficiency and effectiveness in our projects as summarized with quantitative results on Table 2.

Creation process of the classes that our artifact contains (i.e. TCP, UDP, SerialChannel) is not realized by our artifact. However, it is better to have an artifact, which is independent of how its products are created, composed, and represented [29]. Since our artifact serves for different OSs, as a future work, we plan to extend our current artifact with the usage of Abstract Factory design pattern to isolate the user from creation process.

Acknowledgements

The authors would like to thank all their colleagues in ASELSAN for their contributions during the development, and usage of this artifact.

References

[1] A. S. Tanenbaum and M. v. Steen, *Distributed systems : principles and paradigms*, 2nd ed. Upper Saddle RIiver, NJ: Pearson Prentice Hall, 2007.

[2] N. Wang, C. D. Gill, D. C. Schmidt, A. Gokhale, B. Natarajan, J. P. Loyall, *et al.*, "QoS-enabled Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed., ed: Wiley, 2004.

[3] A. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, *et al.*, "Model driven middleware: A new paradigm for developing distributed real-time and embedded systems," *Science of Computer Programming*, vol. 73, pp. 39-58, 9/1/ 2008.

[4] ASELSAN. (2017, 12/02/2017). *ASELSAN*. Available: www.aselsan.com.tr

[5] D. Akdur and V. Garousi, "Model-Driven Engineering in Support of Development, Test and Maintenance of Communication Middleware: An Industrial Case-Study," in *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2015.

[6] IEC/PAS, "Dependability of software products containing reusable components – Guidance for functionality and tests," vol. 62814, ed, 2012.

[7] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *33rd International Conference on Software Engineering*, Waikiki, Honolulu, HI, USA, 2011, pp. 471-480.

[8] A. Gokhale, D. C. Schmidt, B. Natarajan, J. Gray, and N. Wang, "Model Driven Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed., ed: Wiley, 2004.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*: Wiley, 1996.

[10] IBM. (2013, 16/08/2013). *Rational Rhapsody family*. Available: http://www-03.ibm.com/software/products/en/ratirhapfami

[11] W. Chao, L. Hong, G. Zhigang, Y. Min, and Y. Yuhao, "An automatic documentation generator based on model-driven techniques," in *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, 2010, pp. V4-175-V4-179.

[12] D. Parnas, "Precise Documentation: The Key to Better Software," in *The Future of Software Engineering*, S. Nanz, Ed., ed: Springer Berlin Heidelberg, 2011, pp. 125-148.

[13] M. Lindvall, S. Komi-Sirviö, P. Costa, and C. Seaman, "Embedded Software Maintenance," Data and Analysis Center for Software2003.

[14] F. Belli, "Dependability and Software Reuse -- Coupling Them by an Industrial Standard," in *SERE-C '13 Proceedings of the 2013 IEEE Seventh International Conference on Software Security and Reliability Companion* 2013, pp. 145-154.

[15] D. Akdur and Ç. Özdemir, "The Impacts of Reusable and Configurable Modules on Software Quality in Real-Time Embedded Systems: Radar Utility Libraries (In Turkish: Gerçek Zamanlı Gömülü Sistemlerde Yeniden Kullanılabilir ve Yapılandırılabilir Yazılımların Kaliteye Etkisi: Radar Projeleri Destek Kütüphaneleri)," in *8th Turkish National Software Engineering Symposium (In Turkish: Ulusal Yazılım Mühendisliği Sempozyumu (UYMS))*, Cyprus, 2014, pp. 177-186.

[16] P. Mohagheghi, S. Ict, and R. Conradi, "An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product," *ACM Transactions on Software Engineering and Methodology*, vol. 17, pp. 13:1-13:31, 2008.

[17] I. Sommerville, *Software Engineering*: Addison Wesley, 2010.

[18] J. Kramer, "Is abstraction the key to computing?," *Commun. ACM*, vol. 50, pp. 36-42, 2007.

[19] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, "The impact of UML documentation on software maintenance: an experimental evaluation," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 365-381, 2006.

[20] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success

or failure," *Science of Computer Programming*, vol. 89, Part B, pp. 144-161, 2014.

[21] B. P. Douglass, *Real-time UML: Developing Efficient Objects for Embedded Systems*: Addison-Wesley, 2000.

[22] B. P. Douglass, *Real Time UML: Advances in the UML for Real-time Systems*: Addison-Wesley, 2004.

[23] G. M. Nicolescu, P. J., *Model-Based Design for Embedded Systems* CRC Press, 2009.

[24] S. Gerard, J.-P. Babau, and J. Champeau, *Model Driven Engineering for Distributed Real-Time Embedded Systems*: Wiley-IEEE Press, 2010.

[25] D. Akdur, V. Garousi, and O. Demirörs, "Cross-factor analysis of software modeling practices versus practitioner demographics in the embedded software industry," in *6th Mediterranean Conference on Embedded Computing (MECO)*, Montenegro, 2017.

[26] D. Akdur, O. Demirörs, and V. Garousi, "Characterizing the development and usage of diagrams in embedded software systems," in *43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Vienna, Austria, 2017.

[27] J. Davies, J. Gibbons, J. Welch, and E. Crichton, "Model-driven engineering of information systems: 10 years and 1000 versions," *Science of Computer Programming*, vol. 89, Part B, pp. 88-104, 9/1/ 2014.

[28] D. Akdur, O. Demirörs, and V. Garousi. (2015, Technical report of a world-wide survey on software modeling and model-driven engineering in the embedded software industry. [Technical Report]. Available: https://drive.google.com/file/d/0BzPI4c-GGTgoVlAzNDR6Q2I3ZDA/view

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1998.

[30] C. Zhang and D. Budgen, "What Do We Know about the Effectiveness of Software Design Patterns?," *IEEE Transactions on Software Engineering*, vol. 38, pp. 1213-1231, 2012.

[31] H. Mu and S. Jiang, "Design patterns in software development," in *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*, 2011, pp. 322-325.

[32] H. Kun-Yuan, L. Yen-Chih, L. Chi-Hua, and L. Jenq Kuen, "The support of software design patterns for streaming RPC on embedded multicore processors," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, 2008, pp. 263-268.

[33] R. Vincke, S. Van Landschoot, P. Cordemans, J. Peuteman, E. Steegmans, and J. Boydens, "Algorithm Parallelization Using Software Design Patterns, an Embedded Case Study Approach," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, 2013, pp. 470-473.

[34] T. Ovatman and F. Buzluca, "Software Design Pattern Behavior in Shared Memory Multiprocessor Systems," in *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, 2009, pp. 1-4.

[35] J. S. Fant, H. Gomaa, and R. G. Pettit, "Architectural Design Patterns for Flight Software," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, 2011, pp. 97-101.

[36] S. Siegl, K. S. Hielscher, and R. German, "Modeling and Statistical Testing of Real Time Embedded Automotive Systems by Combination of Test Models and Reference Models in MATLAB/Simulink," in *Systems Engineering (ICSEng), 2011 21st International Conference on*, 2011, pp. 180-185.

[37] D. C. Schmidt and S. D. Huston, *C++ Network Programming: Mastering complexity with ACE and patterns*: Addison-Wesley, 2002.

[38] J. Martinez, P. Merino, and A. Salmeron, "Applying MDE Methodologies to Design Communication Protocols for Distributed Systems," in *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*, 2007, pp. 185-190.

[39] F. Fleurey, B. Morin, A. Solberg, and O. Barais, "MDE to Manage Communications with and between Resource-Constrained Systems," in *Model Driven Engineering Languages and Systems*. vol. 6981, J. Whittle, T. Clark, and T. Kühne, Eds., ed: Springer Berlin Heidelberg, 2011, pp. 349-363.

[40] D. Stewart, M. Domeika, S. A. Hissam, S. Hovsmith, J. Ivers, R. Dickson, *et al.*, "Chapter 17 - Multicore Software Development for Embedded Systems: This Chapter draws on Material from the Multicore Programming Practices Guide (MPP) from the Multicore Association," in *Software Engineering for Embedded Systems*, ed Oxford: Newnes, 2013, pp. 563-612.

[41] B. P. Douglass, *Real-Time Design Patterns : robust scalable architecture for Real-time systems*. Boston, MA: Addison-Wesley, 2003.

[42] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*: Addison-Wesley Longman Publishing Co., Inc., 2003.

[43] W. Qiang, X. Jiamou, L. Xuwen, and J. Kebin, "The research and implementation of interfacing based on PCI express," in *Electronic Measurement & Instruments, 2009. ICEMI '09. 9th International Conference on*, 2009, pp. 3-116-3-121.

[44] S. R. Gopalan. (1998). *A Detailed Comparison of CORBA, DCOM, and Java/RMI*. Available: http://my.execpc.com/~gopalan/misc/compare.html

[45] A. S. Krishna, D. C. Schmidt, R. Klefstad, and A. Corsaro, "Real-time CORBA Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed., ed: Wiley, 2004.

[46] R. Schantz and D. C. Schmidt, "Middleware for Distributed Systems," in *Encyclopedia of Computer Science and Engineering*, B. Wah, Ed., ed, 2008.

[47] A. S. Tanenbaum and T. Austin, *Structured Computer Organization*, 6th ed.: Prentice Hall, 2012.