Assessing the Effectiveness of UML Models in Software System Development

Ahmad AbdulQadir AlRababah

Faculty of Computing and Information Technology in Rabigh, Rabigh 21911, KSA. King Abdulaziz University

DOI: https://doi.org/10.56293/IJASR.2024.5703

IJASR 2024
VOLUME 7
ISSUE 1 JANUARY – FEBRUARY **ISSN: 2581-7876**

Abstract: The substantial advancements in microelectronic technology over recent decades have resulted in a progressive expansion of hardware resources. Nevertheless, the utilizable fraction of these resources for actual application realization constitutes a mere fraction of the overall availability, commonly referred to as the design gap. The escalating intricacy of designs, coupled with the imperative of minimizing time-to-market, necessitates the exploration of innovative development methodologies to effectively tackle challenges such as verification, synthesis, and testing within reasonable timeframes. This paper introduces an approach facilitating comprehensive, precise, and object-oriented specifications for hardware/software systems, employing UML 2.0. Furthermore, it details an automated compilation process that transforms these specifications into directly executable applications encompassing both hardware and software modules. Our proposed methodology is implemented through a UML model compiler tailored for reconfigurable architectures, denoted as MOCCA, which conducts validation, mapping, and application synthesis. The proposed methodology is operationalized through the development of a UML model compiler tailored for reconfigurable architectures, named MOCCA. MOCCA not only implements our innovative approach but also performs crucial tasks such as validation, mapping, and application synthesis. This integrated solution aims to bridge the design gap effectively, providing a comprehensive and streamlined framework for the development of hardware/software systems in the face of evolving technological landscapes.

Keywords: Unified Modeling Language (UML), object-oriented specifications, Microelectronic technology, Model Driven Architecture (MDA), System-on-Chip (SoC).

1. Introduction

In recent years, a discernible trend has emerged; indicating a pronounced shift towards the integration of methods, languages, and tools traditionally employed in the software domain for the computer-aided development (CAD) of hardware and mixed hardware/software systems. This trajectory has witnessed a departure from conventional programming-language-centric approaches, exemplified by System C, Spec C, and SPARK. In lieu of these, the adoption of the Unified Modeling Language (UML) has emerged as an increasingly promising avenue for tackling contemporary challenges in hardware/software system development.

Despite the promise of UML-based approaches, it is noteworthy that the majority of current endeavors primarily emphasize software aspects. Critical facets such as design space exploration, hardware/software synthesis, and the rigorous addressing of challenges related to estimation and verification have not received adequate attention in these UML-centric development efforts. This observation underscores the need for a more comprehensive integration of UML in addressing the multifaceted challenges inherent in hardware/software system development. Specifically, there is a pressing need for the augmentation of UML-based methodologies to encompass critical aspects like design space exploration and the intricacies associated with hardware/software synthesis. Furthermore, the refinement of techniques for accurate estimation and verification within the UML framework is imperative to bolster its effectiveness in the context of mixed hardware/software systems. This nuanced integration of UML holds the potential to usher in a more holistic and efficacious paradigm for the computer-aided development of contemporary hardware and mixed hardware/software systems.

Our recent research endeavors are centered on achieving a comprehensive, accurate, and object-oriented specification of computationally intensive applications, leveraging the Unified Modeling Language (UML) and the MOCCA Action Language (MAL). This approach is underpinned by the utilization of the MOCCA-compiler,

Copyright © 2024 IJASR All rights reserved

denoted as the Model Compiler for reconfigurable Architectures, enabling the automated compilation of such specifications into directly executable solutions that seamlessly integrate both hardware and software components. The targeted hardware architectures encompass conventional microprocessors as well as reconfigurable resources such as field-programmable gate arrays (FPGAs). Our development methodology is intricately aligned with model-driven architecture, platform-based design, and hardware/software co-design principles. By amalgamating these approaches, our research seeks to advance the state-of-the-art in the complete and precise specification of computationally intensive applications, providing a pathway for the automated generation of executable solutions across diverse hardware architectures.

This paper introduces an innovative methodology for synthesizing reconfigurable hardware from Unified Modeling Language (UML) models. The proposed approach facilitates the transformation of object-oriented specifications into hardware circuits. The underlying model of computation involves objects communicating through structured messages, wherein fundamental object-oriented principles such as inheritance, polymorphism, encapsulation, and information hiding are accommodated. This approach is versatile, applicable to both stand-alone hardware synthesis and hardware/software co-synthesis environments. Notably, it represents a pioneering advancement by enabling the comprehensive realization of object-oriented specifications, thereby obviating the conventional paradigmatic misalignment inherent in existing object-oriented hardware development efforts across different frameworks, languages, and tools. The subsequent sections of this paper are organized as follows: The initial segment provides foundational insights into the modeling of object-oriented software using UML and offers a concise overview of our development approach, including the relationships between the utilized application models and platform models. Subsequently, the paper delves into the synthesis of software implementations, followed by the presentation of a logical hardware/software interface for objects and components, elucidating their respective life cycles.

2. UML Fundamentals

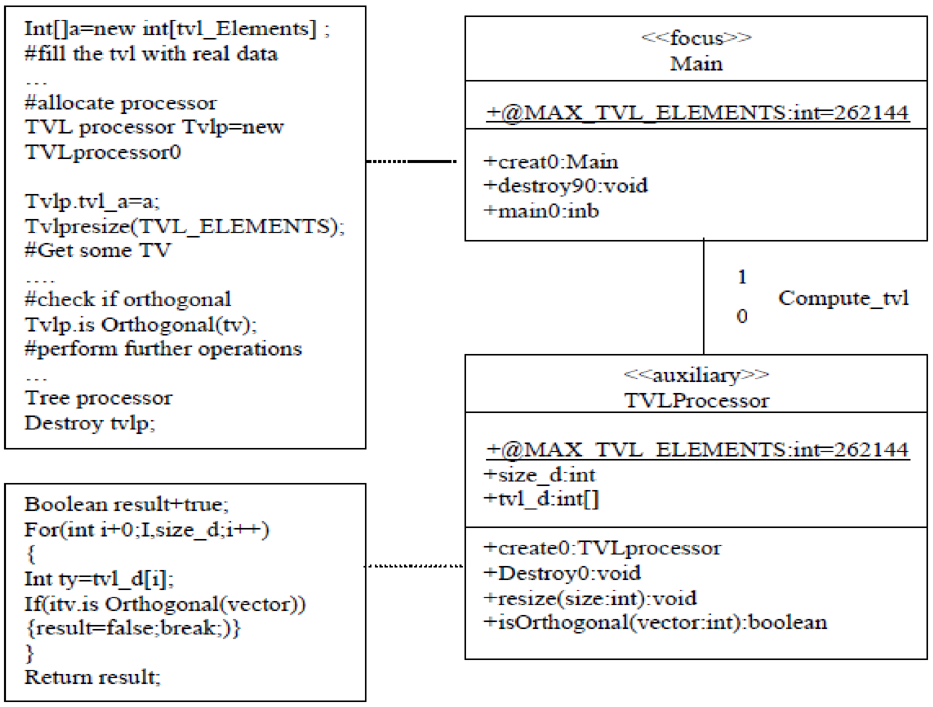
Unified Modeling Language (UML) stands as a comprehensive and universally applicable description language designed to encapsulate diverse facets of object-oriented software. It serves as a versatile tool for representing both the static and dynamic dimensions of software systems. The static attributes of a given system, which encompass its structural aspects, are articulated through various UML diagrams. Among these are class diagrams, component diagrams, and deployment diagrams, each meticulously crafted to convey specific structural details?

Class diagrams provide a visual representation of the classes within a system, illustrating their attributes, relationships, and methods. Component diagrams focus on the organization and dependencies among system components, portraying the high-level architecture. Deployment diagrams, on the other hand, depict the physical deployment of software components across hardware nodes, offering insights into the distribution and configuration of the system.

Complementing the static representation, UML employs behavioral diagrams to capture the dynamic aspects of a system's functionality. These include activity diagrams, sequence diagrams, and state machine diagrams, each serving a distinct purpose in elucidating the system's operational behavior.

In the context of this paper, we specifically utilize UML diagrams that articulate the system structure of a Television Language Processor (TVL-Processor) example, as illustrated in Figure 1. By employing these UML diagrams, we aim to effectively communicate the intricacies of the TVL-Processor system, leveraging the richness and precision that UML affords in representing both static and dynamic elements of object-oriented software. This strategic use of UML ensures a robust and comprehensive depiction of the TVL-Processor's intricacies, facilitating a nuanced understanding of its structure and behavior.

Copyright © 2024 IJASR All rights reserved


```
Int[]a=new int[tvl_Elements] ;
#fill the tvl with real data
...
#allocate processor
TVL processor Tvlp=new
TVLprocessor0

Tvlp.tvl_a=a;
Tvlpresize(TVL_ELEMENTS);
#Get some TV
....
#check if orthogonal
Tvlp.is Orthogonal(tv);
#perform further operations
...
Tree processor
Destroy tvlp;
```

```
Boolean result+true;
For(int i+0;I,size_d;i++)
{
Int ty=tvl_d[i];
If(itv.is Orthogonal(vector))
{result=false;break;)}
}
Return result;
```

Figure 1: Class diagram for Example Design Model

This rudimentary application serves the purpose of scrutinizing the orthogonality of a given set of ternary vectors. The application's behavior is elucidated through meticulously articulated operations specified in the MOCCA action language. A class diagram, employed for modeling the hierarchical structure of classes, serves as a visual representation that encapsulates relationships, including containment, inheritance, associations, and other pertinent connections. In the graphical representation of a class diagram, a rectangular entity symbolizes a class (refer to Fig. 1). This rectangular representation is subdivided into three distinctive sections. The uppermost part accommodates the class name, while the middle section delineates the attributes associated with the class. The lower part comprehensively delineates the operations pertinent to the classes under consideration. This structured depiction within the class diagram facilitates a nuanced comprehension of the interplay between classes, attributes, and operations, providing a comprehensive visual tool for understanding the application's design and functionality.

The association relationship stands as the preeminent and pervasive connection within a class diagram, exemplifying the collaborative interplay between instances of distinct classes. In a paradigmatic illustration, the class denoted as "Main" is intricately associated with the "TVL-Processor" class, underscoring the fundamental nature of associations in delineating the interactions between diverse class entities.

A recurrent relational construct within class diagrams is the generalization relationship. This connection signifies that one class assumes the inheritance of both attributes and operations from another class, thereby establishing a hierarchical inheritance structure. Evidently, in the illustrative depiction provided in Figure 2, the class denoted as "object" serves as the foundational or base class for several derivative classes, namely "Boolean," "bit," "remote," "lime," "string," "char," and another instantiation of the class "object", which encapsulates an array.

The generalization relationship, thus, embodies a pivotal mechanism within the class diagram, elucidating the inheritance hierarchy and propagating shared attributes and operations across classes. This structured representation, inherent in the generalization relationship, contributes significantly to the visual clarity and semantic coherence of the class diagram, providing a sophisticated means for comprehending the intricate relationships and inheritance dynamics within an object-oriented system.

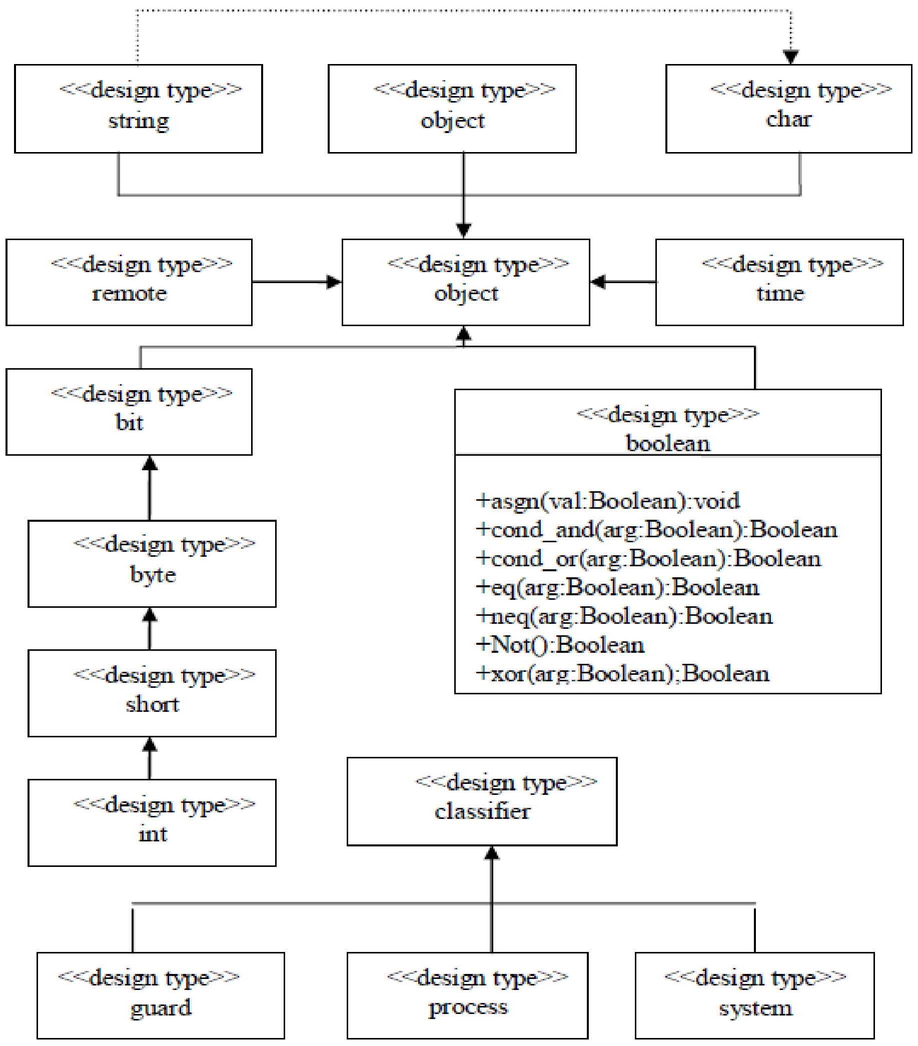


Figure 2: Class Diagram for Design Platform Model

Deployment diagrams serve as visual representations unveiling the physical configuration of a system, delineating the allocation of software components onto specific hardware entities. Central to the diagrammatic portrayal are nodes interconnected by communication paths, as illustrated in Figure 3. Each node embodies a distinct piece of hardware and has the capacity to host particular software elements. Notably, nodes serve as containers or deployment platforms for artifacts, which materialize as tangible representations of software entities, typically taking the form of files.

In Figure 3, for instance, the node denoted as "h0" signifies a microprocessor, actively hosting and executing the runtime artifact "tvlprocessor.exe." An artifact, in a broader sense, is also employed to symbolize a concrete instance of a component. To elucidate, the artifact labeled "HWTestComp.bit" not only represents an instance of the component "HWTestComp" but is also deployed within the FPGA node.

The UML representation employs dotted arrows to signify dependencies, indicating a relationship wherein alterations to the definition of one element may precipitate modifications in the other. The "realize" dependency, specifically, denotes that the source element serves as an implementation of a specification articulated by the target element. This structured and interconnected portrayal in deployment diagrams offers a comprehensive view of the physical arrangement of software components, their corresponding hardware hosts, and the intricate dependencies that underpin their operational cohesion within a system.

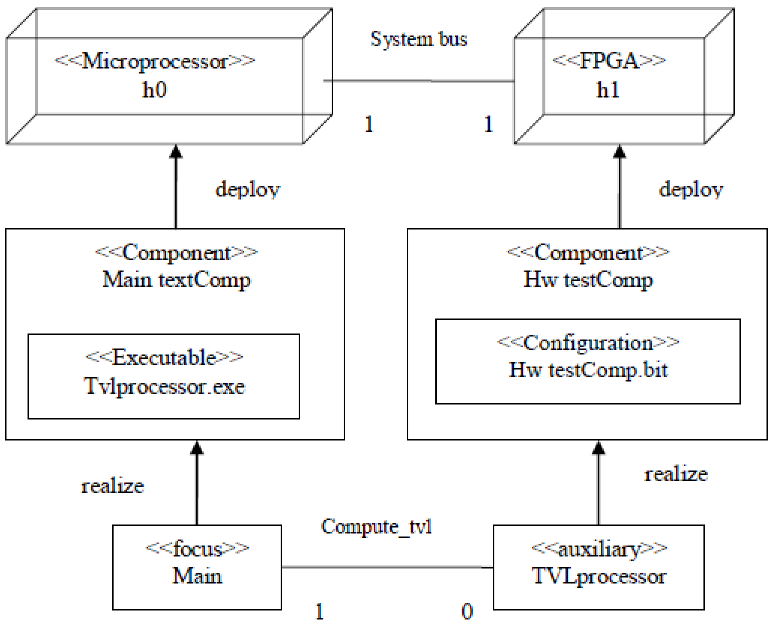


Figure 3: Implementation and Development Model for example design model

3. Development Methodology

Within this section, a concise overview of the overarching development approach is presented. Figure 4 provides a visual representation of the fundamental activities and artifacts intrinsic to our development methodology. The delineated approach seamlessly integrates the established methodology of software/hardware co-design into the overarching framework of Model-Driven Architecture (MDA). The strategic alignment with MDA signifies a methodological orientation that prioritizes the utilization of models as central artifacts throughout the development lifecycle. MDA emphasizes abstraction, encapsulation, and the systematic refinement of models to generate concrete implementations, thereby fostering a modular and systematic approach to software and hardware development.

Figure 4 serves as a graphical elucidation, encapsulating the core activities and artifacts entailed in the development process. This symbiotic integration of software/hardware co-design and MDA establishes a cohesive and structured foundation for the systematic realization of complex systems, where the iterative refinement of models plays a pivotal role in achieving both precision and efficiency in the development endeavor.

In our methodology, the conceptualization of the application design takes the form of an object-oriented system-level Unified Modeling Language (UML) model, as illustrated in Figure 1. Serving as the foundational framework for the design model is the design platform, which is meticulously characterized by a set of types and constraints, as exemplified in Figure 2.

The instantiation of functions, structure, and behavioral attributes of the objects is encapsulated within a platform-independent design model (PIM). This PIM operates as a comprehensive representation that abstracts from platform-specific intricacies, ensuring a level of generality and modularity conducive to later stages of the design process.

Concurrently, the target platform model (TPM) assumes prominence, delineating all requisite information for platform mapping and synthesis. The TPM, articulated as a UML model, explicitly defines the architecture of the target hardware. This architectural description encompasses hardware nodes, such as reconfigurable devices and microprocessors, the interconnecting communication paths between these nodes, and the inherent capabilities that these hardware components furnish for the implementation of the application, as depicted in Figure 3. By leveraging UML as the modeling language, this approach facilitates a unified and coherent representation, bridging the conceptual design phase with the intricacies of platform-specific implementation considerations.

Copyright © 2024 IJASR All rights reserved

The culmination of our development process involves the meticulous mapping of the application onto the structural framework of the target hardware system, effecting the transformation from a Platform-Independent Model (PIM) to a Platform-Specific Model (PSM). This critical phase entails the adaptation of the PIM to align with the distinctive characteristics and resources of the specific target platform. In instances where the implementation necessitates supplementary support from an operating system, particularly for tasks such as inter-object synchronization and communication, the pertinent elements within the PIM are intricately bound to the resources furnished by the corresponding devices on the target platform.

The conclusive stage of our developmental trajectory is synthesis. Commencing with a platform-specific model that encapsulates the intricacies of our application, we systematically embark on the transformative process of synthesizing this model into a functional, ready-to-run implementation. The synthesis operation entails the generation of executable software modules for functionalities designated to run on microprocessors. Simultaneously, for functionalities earmarked for execution on reconfigurable logic or hardware modules, the requisite components are systematically generated.

This synthesis process, informed by the specifics of the target hardware system and the nuanced design choices encapsulated in the PIM and PSM, culminates in a cohesive and executable implementation. The bifurcated synthesis approach, addressing both software and hardware modules, underscores the versatility and adaptability of our development methodology in accommodating diverse computational requirements.

4. Platforms and Models

The underpinning of application development resides in the utilization of distinct platforms, with divergent platforms specifically designated for design, implementation, and deployment purposes. This strategic partitioning of development concerns serves to establish a rigorous separation, thereby facilitating enhanced efficacy in validation, portability, adaptability, and the facilitation of component reuse. The conceptual framework of platforms has been pervasive throughout the evolution of software and hardware development. However, it is noteworthy that platforms, conventionally, are tacitly encapsulated within language reference manuals, libraries, and tools, presenting a hindrance to their seamless and automated interpretation by computational systems.

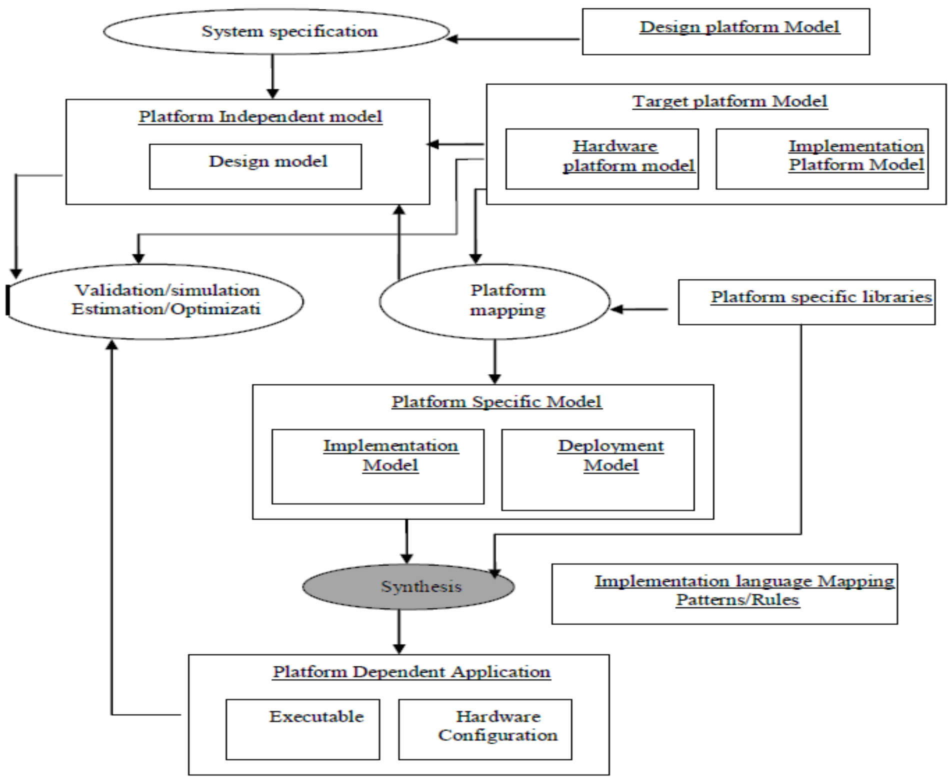


Figure 4: Development Methodology-activities and Artifacts

Copyright © 2024 IJASR All rights reserved

Platforms serve as foundational sets of assumptions that underpin every developmental endeavor. In the elucidated approach, these assumptions are expressly articulated through platform models, with each platform being meticulously delineated by a dedicated Unified Modeling Language (UML) model. These platform models serve the crucial function of abstracting from the intricacies inherent to the specified platform, yet encapsulating sufficient information to obviate the need for repetitive iterations within the design flow. Effectively, they establish the groundwork for defining application models, each of which encapsulates specific facets of the evolving system.

The symbiotic interplay between platform models and application models is illustrated in Figure 5. This intricate relationship denotes the pivotal role played by platform models in shaping and informing the subsequent instantiation of application models, which, in turn, encapsulate distinct aspects of the developmental landscape. The explicit specification and interconnection of these models not only streamline the design process but also fortify the systematic coherence and fidelity between the foundational assumptions and the diverse facets of the evolving system under development.

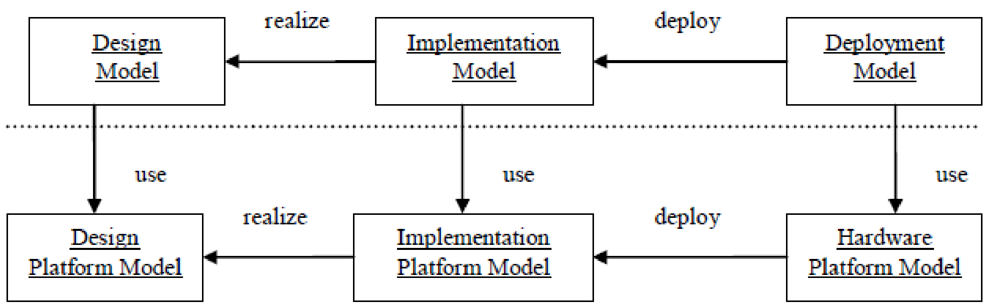


Figure 5: Relationships between models

The platform models delineated beneath the horizontal demarcation ascertain the domain within which applications may be formulated, contingent upon the specific platforms in question. Above this demarcation, models depicting concrete applications, thereby signifying distinct points within the application space, are presented. It is customary for platform models to be shared among multiple application models, fostering a modular and reusable framework wherein the foundational characteristics of the platform are leveraged consistently across diverse applications. This hierarchical relationship underscores the abstraction provided by platform models, which define the overarching conditions and constraints within which a spectrum of applications may be developed. The demarcation between platform and application models ensures a systematic and scalable approach to the development of varied applications within the defined platform ecosystem.

5. Software Implementation

In the preceding section, we provided a succinct overview of our development methodology and elucidated the interplay between application and platform models. In this section, we expound upon an object-oriented synthesis approach for the generation of software from Unified Modeling Language (UML) models. The instantiation of classes within the implementation model, intended for deployment on microprocessor nodes, is directly realized in the C++ programming language.

Facilitating communication between local and remote objects is orchestrated through the deployment of local proxy objects. For each remote object accessed by a local object, a corresponding proxy is instantiated locally. This proxy serves as an intermediary, encapsulating the communication mechanism and thereby obviating the necessity for shared address spaces among the application's objects. Explicit modeling of the proxy is incorporated within the implementation platform model, classifying it as a remote type, as depicted in Figure 2. This meticulous modeling ensures that the model compiler can derive high-quality estimates pertaining to the characteristics of distributed applications.

Objects instantiated in reconfigurable hardware are subject to management by a specialized service known as the RTR-Manager. This managerial entity encapsulates the idiosyncrasies of reconfigurable hardware, encompassing aspects such as reconfiguration modes, input/output functions, and communication protocols. A primary function

Copyright © 2024 IJASR All rights reserved

of the RTR-Manager is the processing of application requests pertaining to the instantiation and termination of hardware objects. Notably, hardware objects are dynamically created and destroyed in response to demand. Applications seeking the instantiation of a hardware object convey their request to the RTR-Manager, specifying the object type. This orchestrated approach ensures a dynamic and responsive instantiation and destruction mechanism tailored to the requirements of the reconfigurable hardware under consideration.

The RTR-Manager undertakes the task of scrutinizing the extant bit streams for a fitting object, subsequently furnishing its corresponding proxy to the application. In the event that none of the extant bit streams currently hosts an object of the sought-after type, the RTR-Manager dynamically initiates the instantiation of a pertinent bit stream to accommodate the required object.

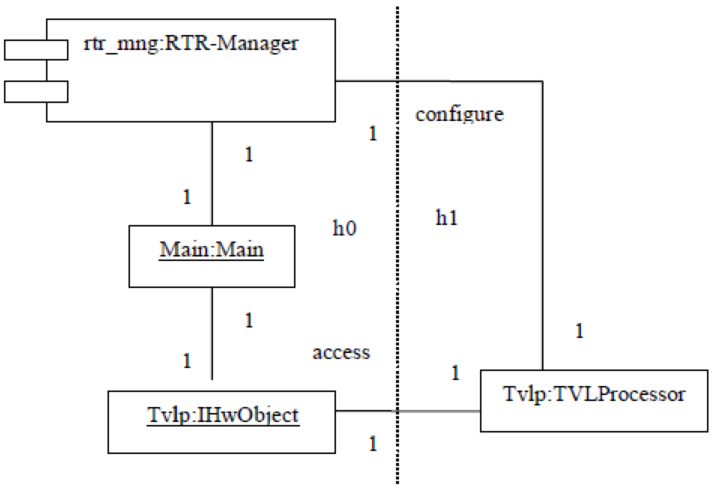


Figure 6: Software Architecture of TVL processor

Figure 6 delineates the fundamental architecture of the TVL-Processor exemplar. The instantiation denoted as "main" belonging to the class "Main," alongside a proxy for a hardware object, finds implementation in the software domain. Contrarily, the tangible manifestation of the "TVL-Processor" class is effectuated through the utilization of reconfigurable resources. The interaction with the hardware object, residing in reconfigurable resources, is mediated through a specialized proxy, systematically provided to the application by an instance of the RTR-Manager service.

Proxies, endowed with the capability to serve as a direct conduit for accessing the corresponding hardware objects within the software implementation, furnish a straightforward yet expeditious mechanism. Alternatively, proxies may be enveloped by software implementations of the hardware object classes. Upon the instantiation of a software object, an endeavor is made to instantiate its hardware counterpart. In the event of a successful instantiation, the hardware object is employed; conversely, in cases of failure, the software object seamlessly transitions to the software implementation. This approach additionally accommodates a seamless transition between hardware and software objects, thereby offering flexibility and adaptability within the developmental paradigm. The automation of such implementations is facilitated by advanced model compilers.

6. Hardware / Software Interface

The direct instantiation of components, classes, and features is inherently convenient and exhibits straightforwardness, attributes well-acknowledged within the realm of software engineering. However, when transposing this approach to hardware implementations, a distinct set of challenges emerges. These challenges, unique to the hardware context, are elucidated as follows:

Dynamic Object Instantiation/Destruction. Owing to the static nature of partially reconfigurable hardware, the expedient instantiation and destruction of hardware objects pose efficiency challenges. The instantiation of classes per reconfigurable device is deemed prohibitively costly, both in terms of the substantial utilization of logic resources and the extensive time required for reconfiguration.

Polymorphic Features. Polymorphism stands as a pivotal attribute within the domain of object-oriented specifications, and its direct accommodation in hardware implementations is imperative. However, prevailing methodologies tend to eschew polymorphism, resorting to the prohibition of inheritance or the overriding of behaviors as a means of circumventing its complexities.

The establishment of autonomous communication among Objects necessitates their ability to engage in interactions irrespective of their specific materialization. Within amalgamated configurations involving software and hardware components, a uniform and singular mechanism for the exchange of messages remains absent. The ensuing sections expound upon the intricate mapping procedures that facilitate the transition from abstract implementation models to the tangible realm of hardware and software implementations.

The hardware/software interface inherent in object-oriented implementations integrated with reconfigurable hardware delineates the life cycle and access mechanisms governing objects and components instantiated within reconfigurable hardware. This interface manifests itself through both logical and physical dimensions. The logical aspect of the hardware/software interface can be instantiated through diverse physical realizations, with the specific implementation contingent upon the targeted platform and the model compiler employed in the process.

Object Life Cycle Disparities. The life cycle dynamics of hardware objects diverge from those governing software objects, driven by imperatives of operational efficiency. To mitigate resource-intensive reconfigurations, the strategy involves maximizing the reuse of hardware objects. Commencing in the state denoted as OBJ_UNBOUND, each hardware object, initially devoid of any association with hardware resources, exists solely as a template within a component actualized by a specific hardware configuration context. Upon instantiation of the component, its configuration is load
ed into the pertinent device, prompting a transition of all constituent objects to the OBJ_BOUND state, signifying their binding to physical hardware resources.

Objects residing in the OBJ_BOUND state are potentially allocable by the application. Following such allocation, the corresponding objects progress to the state of OBJ_ALLOCATED. In the event of an object's liberation, it undergoes an interim transition to the OBJ_UNBOUND state, suggesting its release, and potentially leading to the object's destruction. Concurrently, any enclosed objects revert to the OBJ_UNBOUND state, as depicted in Figure 7.

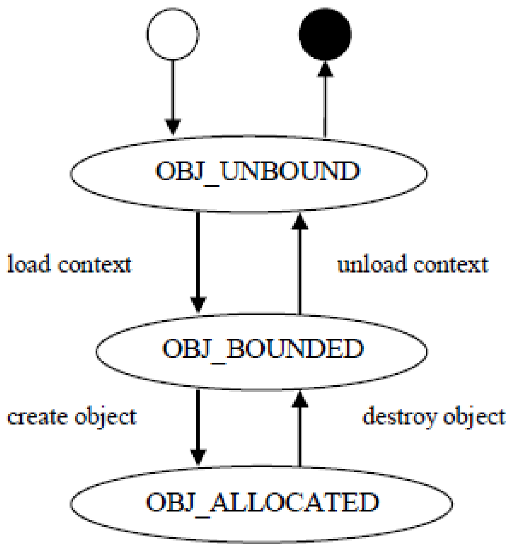


Figure 7: Object Life-cycle

Object Interface Dynamics. The remarkable strides witnessed in the microelectronic industry over recent decades have yielded an ever-expanding reservoir of hardware resources. Notwithstanding this burgeoning availability, the subset of resources practically applicable for application realization constitutes a minute fraction—a phenomenon

Copyright © 2024 IJASR All rights reserved

recognized as the design gap. The escalation in design intricacies, coupled with the imperative of expeditious time-to-market deployment, necessitates the exploration of innovative developmental paradigms. Such paradigms are imperative to effectively contend with challenges encompassing verification, synthesis, and testing, all while ensuring a judicious allocation of effort.

Control Interface and Object Characteristics. The control interface serves as a critical conduit enabling the identification, typing, and access to objects within their designated spatial domain. In this context, objects are distinctly identified in their object space, with the object's identifier (ID) representing its address, a parameter established during the initialization phase. The necessity of including this ID field is contingent upon whether explicit delineation of the object's address is requisite within the object interface. Simultaneously, the type field encapsulates the dynamic type of the object, a determinant factor in the selection of suitable implementations, particularly with respect to polymorphic features. The mandatory inclusion of the type field is contingent upon the object's potentiality for assuming diverse dynamic types. Furthermore, the message fields play a pivotal role in uniquely identifying the nature of services accessed via messages directed at the object. The execution of services in response to such messages may intricately hinge upon the dynamic type of the object. The transmission of message parameters is facilitated through the data interface, which, in turn, allows access to the object's state and facilitates the exchange of input/output parameters between objects. The encompassing interface comprehensively incorporates the public state of the object, along with parameters integral to publicly accessible services. The exception interface, reflective of exceptional conditions within the object, exposes details contingent upon the object's prescribed exception-handling mechanisms. This disclosure includes information regarding the position and type of exceptions, thereby enabling other objects to react judiciously.

Object Access Mechanisms. The delineated object interface affords the capability for the instantiation, destruction, and access to services offered by hardware objects. The mechanics governing object access are explicitly articulated through the object interfaces, each of which comprises a control interface, a data interface, and an exception interface, as delineated in Figure 8.

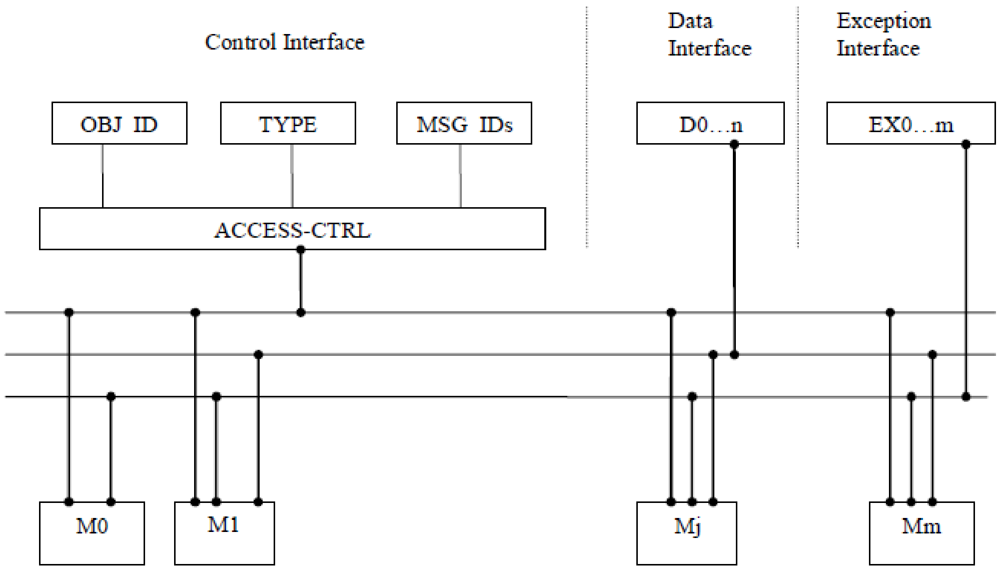


Figure 8: Hardware Object Interface

7. Conclusion

This article introduces a pioneering methodology for UML-based application development tailored for reconfigurable architectures. The approach seamlessly integrates fundamental principles of model-driven architecture, platform-based design, and hardware/software co-design. We have successfully demonstrated the utility of UML in the development of a diverse array of System-on-Chip (SoC) applications. As a case in point, we have provided a concrete illustration through the implementation of a straightforward application designed to assess

Copyright © 2024 IJASR All rights reserved

the orthogonality of a Ternary Vector List. The presented application serves as a compelling exemplar showcasing the transformation of platform-independent UML design models into the definitive realizations of hardware/software modules. This paradigm not only underscores the versatility of UML in the development of SoC applications but also highlights its efficacy in bridging the conceptualization phase with the eventual implementation of intricate hardware/software modules. The mentioned application serves as a practical demonstration of the process involved in translating abstract, platform-independent UML (Unified Modeling Language) design models into concrete and executable implementations of hardware and software modules. The term "platform-independent" implies that the UML design models are not tied to any specific hardware or software platform, allowing for flexibility in their representation. The application, which involves checking the orthogonality of a Ternary Vector List, showcases how UML can be effectively utilized throughout the development cycle, from initial design to the eventual realization of hardware and software components. This demonstration underscores the capability of UML as a modeling language to bridge the gap between conceptual design and the tangible implementation of complex systems on reconfigurable architectures.

References

1. Ahmad AlRababah, "Implementation of Software Systems Packages in Visual Internal Structures", Journal of Theoretical and Applied Information Technology, Volume 95, Issue 19 (2017), Pages: 5237-5244.
2. Chintalapati, Shireesha, and M.V. Raghunadh. "Automated attendance management system based on face recognition algorithms." International Conference on Computational Intelligence and Computing Research. IEEE, 2015.
3. Ahmad AlRababah "Implementations of Hybrid FPGA Microwave Format Extension as a Control Device", IJCSNS International Journal of Computer Science and Network Security, VOL.18 No.11, November 2018.
4. Jha, Abhishek. "Classroom attendance system using facial recognition system." The International Journal of Mathematics, Science, Technology and Management, 2014.
5. Ahmad AlRababah "Watermarking implementation on digital images and electronic signatures", International Journal of Advanced and Applied Sciences, Volume 4, Issue 10 (October 2017), Pages: 160-164.
6. Riya, G. Lakshmi, et al. "Implementation of attendance management system using SMART-FR." International Journal of Advance Research Computer and Communication Engineering, 2015.
7. Ahmad AlRababah. "A New Model of Information Systems Efficiency based on Key Performance Indicator (KPI)" (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 8, No. 3, 2017.
8. Akash G, Rupali B and Shobhana S. "SDLC (Software Development Life Cycle)". Published 2014. https://www.slideshare.net/akash250690/sdlc-models-38873234.
9. A. A. AlRababah, "Neural networks precision in technical vision sys-tems," IJCSNS, vol. 20, no. 3, p. 29, 2020.
10. K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.
11. M. Yousef, K. F. Hussain, and U. S. Mohammed, "Accurate, data-efficient, unconstrained text recognition with convolutional neural net-works," Pattern Recognition, vol. 108, p. 107482, 2020.
12. Israa Al-Barazanchi, Aparna Murthy, Ahmad Abdul Qadir Al Rababah, Ghadeer Khader, Haider Rasheed Abdulshaheed, Hafiz Tayyab Rauf, Elika Daghighi, Yitong Niu. "Blockchain Technology - Based Solutions for IOT Security" IJCSM: Iraqi Journal for Computer Science and Mathematics, vol. 3, no. 1, Jan. 2022
13. L. Biewald, "Experiment tracking with weights and biases," 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com
14. Ahmad AlRababah "Assurance Quality and Efficiency in Corporate Information Systems", IJCSNS International Journal of Computer Science and Network Security, VOL.19 No.4, April 2019.
15. T. Jayalakshmi and A. Santhakumaran, "Statistical normalization and back propagation for classification," International Journal of Computer Theory and Engineering, vol. 3, no. 1, pp. 1793–8201, 2011.
16. Ahmad AlRababah "Problems Solving of Cell Subscribers based on Expert Systems Neural Networks" International Journal of Advanced Computer Science and Applications (IJACSA), 10(12), 2019.
17. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

Copyright © 2024 IJASR All rights reserved

18. 14. S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.
19. Ahmad AlRababah , "DIGITAL IMAGE ENCRYPTION IMPLEMENTATIONS BASED ON AES ALGORITHM", VAWKUM Transactions on Computer Sciences, Volume 13, Number 1, May-June , 2017, Pages: 1-9.
20. Pedrycz, Witold. Granular computing: analysis and design of intelligent systems. CRC press. Published 2018.
21. Reema Mrayyan, Ahmad AlRababah, "Debugging of Parallel Programs using Distributed Cooperating Components". IJCSNS International Journal of Computer Science and Network Security, VOL.21 No.12, December 2021.
22. Antoniou, Andreas. Digital filters: analysis, design, and signal processing applications. McGraw-Hill Education. Published 2018
23. Ahmad AlRababah, "Neural Networks Precision in Technical Vision Systems" IJCSNS International Journal of Computer Science and Network Security, VOL.20 No.3, March 2020.
24. Atakishiyev, Shahin, et al. "A multi-component framework for the analysis and design of explainable artificial intelligence." Published 2020.
25. 21. Vishal, " Python SQLite tutorial using sqlite3" Updated in 2021.
26. Ahmad AlRababah, Ahmad Alzahrani. "Software Maintenance Model through the Development Distinct Stages", IJCSNS International Journal of Computer Science and Network Security, VOL.19 No.2, February 2019.
27. Thomas Hamilton, "What is Software Testing? Definition, Basics & Types in Software Engineering". Published 2021.
28. Lalbihari Barik, Ahmad AbdulQadir AlRababah, Yasser Difulah Al-Otaibi. "Enhancing Educational Data Mining based ICT Competency among e-Learning Tutors using Statistical Classifier" International Journal of Advanced Computer Science and Applications (IJACSA), Volume 11 Issue 3 March 2020.
29. Ahmad AlRababah, Bandar Ali Alghamdi. "Information Protection Method in Distributed Computer Networks Based on Routing Algorithms" IJCSNS International Journal of Computer Science and Network Security, VOL.19 No.2, February 2019.
30. Ahmad AlRababah. "Data Flows Management and Control in Computer Networks", (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 9, No. 11, 2018.
31. Ahmad AlRababah. "On the associative memory utilization in English- Arabic natural language processing", International Journal of Advanced and Applied Sciences, Volume 4, Issue 8 (August 2017), Pages: 14-18.

Copyright © 2024 IJASR All rights reserved