Can UML be a System-Level Language for Embedded Software? [*]

João M. Fernandes, Ricardo J. Machado
Dep. Informática & Dep. Sistemas de Informação
Universidade do Minho, Braga, Portugal

Abstract: The main purpose of this paper is to discuss if the Unified Modeling Language (UML) can be used as a system-level language (SLL) for specifying embedded systems, in co-design environments. The requirements that a language has to fulfil to be considered as an SLL are presented and the advantages and disadvantages of using UML as an SLL are also indicated. The contribution of this paper consists on the explicit discussion of the key issues that must be taken into account when deciding if UML is to be used in a project as an SLL for embedded software.

1. INTRODUCTION

The discussion on the “best” system-level language is a key topic on the area of embedded software development. Among the alternatives, the following are generally identified: C++, Java, domain-specific languages and pure semantics. This last choice is not a language, in the proper sense, but is typically introduced taking into consideration that the language (i.e. its syntax) is not an issue, and what really matters is its semantics.

Although the designers have several alternatives to choose from, a vast majority of people still use C/C++ as the languages for solving their co-design problems and find them good SLLs. C/C++ are indeed suitable solutions for implementing embedded systems, but those languages are not adequate for system-level modelling. Although C++ is an object-oriented extension of C, both present the same basic characteristics. This implies that

[*] Research funded by FCT and FEDER under project *METHODES* (POSI/37334/CHS/2001).

C/C++ are languages near the hardware, which is a good characteristic for achieving strong predictability on execution time (a fundamental issue for real-time systems), but also that they lack some of the characteristics that an SLL should present, which, are hierarchy, concurrency, programming constructs, abstract communication, synchronization mechanisms, exception handling, structural representation and state-based constructs [13].

The analysis presented in this paper is especially oriented for heterogeneous environments, where the hardware and the software components are equally treated, during the analysis phase, namely in what concerns the modelling and specification of behavioural and non-functional requirements. This does not mean, however, that the ideas presented here are limited to that specific field. Several concepts and arguments presented here can also be applied to systems that can be classified as being complex and with strong non-functional constraints. Typical hardware-based solutions, where systems-on-a-chip are the most constringent instances, are not being considered. Instead, software-based systems with strong constrictions, such as real-time, fault-tolerance and explicit concurrency, are the main kind of systems to consider. The paper does not address how to technically use UML for the specification of embedded systems, since that topic is well covered in the literature. For example, in [12] a UML profile called "Embedded UML" is presented and several other groups have made proposals for developing embedded systems with UML [3, 7, 16]. The opinions expressed in this paper about the system-level capabilities of UML are based on the experience gained with its application in real industrial projects [5, 6].

2. EMBEDDED SYSTEMS DESIGN

Until recently, researchers have largely ignored embedded systems development since, as a scientific problem, it was small and not interesting. This reality has changed for many different factors, and now computer scientists are beginning to pay more attention to the embedded arena [9].

Typically, embedded systems have specialized functionality, incorporate microprocessors and have a limited capacity of memory. To meet size and performance requirements, designers usually use a real-time operating system (RTOS) and proprietary development tools, well-tuned for meeting the devices' memory limitations.

In the past, embedded systems were developed in assembly languages. Later, due to more complex functionality, some companies turned to higher-level languages (HLLs) like C and C++. HLLs make it easier to develop the systems, but they still present problems, due to their inherent complexity, which implies long schedules and high non-recurring engineering costs.

To exacerbate these problems, there were a greater number of target operating systems and processors, sometimes even within the same product families. Manufacturers faced enormous competitive pressures, and were asked to develop their products in a shorter time.

Nowadays, embedded systems are networked and distributed and, more importantly, consumers demand more complex functionality, which greatly increases software complexity. As a consequence, these systems can no longer be designed as was done traditionally, and new approaches and new languages are required. This implies that describing and modelling a modern embedded system requires an SLL.

UML is one possible solution to this problem, since it promotes a more open, standard-based pre-implementation development environment, which would lower costs and speed development.

3. SYSTEM-LEVEL IN EMBEDDED DESIGN

The system-level is generally described as the abstraction level where the differences between hardware and software are minimal. At this level, the entire system is looked at as a set of cooperating subsystems [15]. This represents a big advantage for real-time embedded systems development, because it allows the system to be specified with a unified (homogeneous) representation, and makes co-design an effective approach for developing heterogeneous implementations.

Since it is quite obvious that traditional languages, especially procedural HLLs and HDLs (Hardware Description Languages), are not able to cope with the ever increasing complexity of embedded systems, a race for defining “the” SLL for co-design of embedded systems is emerging. Among the several alternatives for winning that race, the following ones seem strong and firm candidates: ANSI C/C++, SystemC, Java, Superlog, and Rosetta.

It is not uncommon to mix concepts of using a language for specification (what to design) and using it for implementation (how to design). When referring to SLLs, it should be highlighted that its main usage, within the design flow, is based on a specification-oriented approach, however, it must also allow the introduction of design decisions, by syntactic inscription of refinement tags, to semantically support the (semi-)automatic implementation of the system.

Although there are a variety of different opinions, visions and (commercial and scientific) motivations, with respect to SLLs, as the previous enumeration suggests, it is possible to describe a generic set of requirements that the co-design community accepts more or less consensually for an SLL [1]:

– **Modelling:** An SLL must allow the software and the hardware components of a system to be collectively developed (i.e. co-specified and functionally co-refined), in such a way that the system as a whole can be easily perceived by the project members. Ideally, an SLL should be able to treat all the design space, supporting the semantical specification of the non-functional requirements, which may be provided by different technological areas.
– **Implementation:** An SLL must give an effective support to the system's implementation, based on automatic (or at least, semi-automatic) refinements, to feed synthesizable HDLs and HLLs, in order to justify a co-design approach at the system-level. As a consequence, the complexity can be coped, but, more importantly, the development time is reduced and a guarantee can be given with respect to the implementation of the user's requirements (models' continuity).
– **Simulation:** An SLL must be able to support (and be supported by) powerful simulation environments, where the designed system may be analysed and experimented in relation not only to its functional behaviour but also to its expected performance. The executability of an SLL is a vital characteristic to facilitate the requirements' capture and validation.

In addition to the 3 previous points, there are many others that may be considered. Thus, an SLL should: (i) allow the explicit (or implicit) description of concurrency; (ii) possess a well-defined semantic; (iii) be sufficiently appealing and advantageous to be naturally adopted by designers; (iv) be supported by user-friendly tools; and (v) ensure a reduced learning curve. Nevertheless, they still need a mature decision, since some are pure intentions and others are not at all possible to be satisfied at the moment.

It is not expected that the migration to the system-level with relation to language issues will be fulfilled by SLLs. It is admissible, at least during a transitory phase, to use other languages that may be helpful to describe functionalities not within the scope of the SLLs available at a precise moment. Apart from defining efficient SLLs, it is important to conceive development methodologies to support the design at the system-level. This implies the selection of the various languages to be used, the definition of the development phases, and the relation amongst languages and phases.

Thus, the main question is how to obtain a system-level co-design environment to support the modelling of embedded systems and to assist their semi-automatic implementation. This must be made in such a way that:
– the models may be iteratively reified until the final implementation is obtained, without the need to manually perform macro-refinements, with the transparent reuse of pre-designed hardware target architectures and software modules;

– the activities of the different project members that are involved in complex projects are properly integrated.

4. UML FOR SYSTEM-LEVEL

UML is a general-purpose modelling language for specifying and visualizing the artefacts of computer-based systems, as well as for business modelling and other non-software systems [2]. UML is a standard language for defining and designing software systems, and is being progressively accepted as a language in industrial environments. UML is meant to be used universally for the modelling of systems, including automatic control applications with both hardware and software components, so it seems an adequate choice for embedded systems.

Although UML does not guarantee project success, it may improve many related topics. For example, it substantially decreases the cost of training, when there is the need to make changes in projects or organizations. It also provides the opportunity for new integration among tools, processes and domains. Finally, UML enables designers to focus on delivering business value and provides them the tools and techniques to accomplish this.

4.1 Advantages of using UML

4.1.1 Standard

UML is a multiple-view and graphical notation that presents a variety of diagrams for different modelling purposes. Although the novice UML user can get confused with all these possibilities, it is possible and desirable to choose the important diagrams for a specific application field. One of the main advantages of using UML is that it is a standard. UML is an OMG standard and is expected to become an ISO standard very soon [8]. Being a standard implies that in the near future it is likely that every TI professional will understand it, so it will be widely accepted. This also implies that several computer tools will be produced for simplifying the tasks of drawing the diagrams and for automatically obtaining implementation code.

4.1.2 Communication with the customer

UML is inherently a graphical language. Graphical languages are quite important for promoting the communication between the system's designers and customers. If the communication is not established in a proper way, the designers are not sure that they are building the right system, even if they know how to build the systems right.

Usually, customers have some special interest in the application, but they are not supposed to, although they can, be aware of the technical problems associated with the system. Additionally, designers are expected to be competent in technical matters, but usually it is unlikely that they are experts in every field of application. Thus, to be effective communication between designers and customers must use a notation that is useful for both of them.

If specifications are intended to serve as a communication medium among customers and designers, using graphical notations is essential, as long as they are clear and intuitive (to be created, modified, and inspected by both customers and designers), and also precise and rigorous (to be validated, simulated or analysed by computers).

UML is a valid alternative for this purpose, since it is graphical and not too complicated, but, at the same time, precise. Dialoguing with the customers in C is not possible, at least generally speaking, and communicating in a natural language, although extremely easy, is also not a proper solution, since it introduces too many ambiguities.

4.1.3 Object-oriented modelling

UML is perfectly suited for specifying object-oriented (OO) systems, since it includes several diagrams for that modelling paradigm. Although many embedded systems are still implemented with non-OO languages, the great majority is already developed with OO techniques and in the future it is expected that an even greater majority will use OO principles and languages.

A methodology to system development based on the operational approach is essential to guarantee that complex systems can be addressed. The main idea of this approach is based on an executable specification that evolves through transformational refinements to obtain the final implementation. Object-oriented models are expected to fully address the above requirement, since they allow the easy refinement of application-domain objects during the whole process.

4.1.4 Platform independence

Specifying a system in UML can be absolutely platform-independent, since the specification can be reused for different target architectures, different technologies, different environments, and other non-functional requirements. This is possible because, during the analysis and design phases, UML supports views that can be reified without early introducing undesired implementation decisions, allowing the specification to preserve its system-level nature, until the final implementation synthesis steps.

4.1.5 Automatic code generation

Being an OO notation, the structural and behavioural views of UML can be “easily” transformed into code. UML has the potential to be automatically transformed into any language, being it OO or not. There are some tools that give support to this automatic code generation task, which imply that we are near to reach the point where the specification is the implementation.

In contrast to the situation where the designers specify the system in the final implementation language, using UML and automatic code generation tools allows the system to be converted into different languages. This may be a strong advantage, allowing the same specification to give origin to different implementations for different purposes or for different architectures. The existence of code generators is a key issue to allow different hardware-software partitions to be obtained from the same unbiased specification. In the authors' co-design approach [10], the code generation allows the usage of the software parts with different pre-designed hardware target architectures. The main point is to generate implementation code only for the software parts and not for the hardware target architectures being used. Thus, for this possibility to be real, it is absolutely necessary to model both parts at the system-level.

Generating code from UML may result in problems, if some points are forgotten. Generally, UML is missing implementation details, so it is not easy to perform implementation specific optimisations (for size or speed) from a given UML specification. If the generated code is not good for the purpose in hand, the designer has to write code for implementation. To do this, generated code must be easy to read in order to improve it manually.

4.1.6 Extensions

UML can be extended, since it was elaborated with that particular purpose. This means that UML is not restricted to its original aims (specification and visualization), but that it can be used to other purposes, if the extension mechanism is properly used. Extensions in UML are achieved through stereotypes, that augment the semantics of the meta-model.

There are several proposals to extend UML to support the modelling of embedded systems. *Real-Time UML* [3] is one of the most popular, since it treats all the development phases (analysis, design and implementation) in a simple way. It is worth mentioning that Real-Time UML presents code in C++ that was obtained after modelling the systems in UML.

4.2 Disadvantages of using UML

4.2.1 Number of diagrams

UML is a multiple-view syntactic meta-model, which means that it defines many different diagrams, each one covering a particular modelling perspective of the system. By one hand, this is an advantage, since it allows the designers to specify the aspects they find important for a specific purpose, without imposing a particular development process model. By another hand, this may be a disadvantage, since the diagrams are interrelated, although not formally, which means that inconsistencies can be introduced in the system specification.

Another related problem lies on the fact that there are different diagrams for similar purposes. For example, use case, collaboration, sequence, statecharts and activity diagrams are all used for describing behavioural perspectives of a given system. Although these five diagrams handle different behavioural aspects, this may be confusing for some designers.

4.2.2 Not precise semantics

UML is not a formal notation, i.e., it has not a well-defined semantics. UML is a semi-formal language, because it has a formal syntax, but its semantics is not formal. This fact may impose different interpretations on the semantics, which implies that a diagram may not be equally interpreted by two different designers. Some authors have proposed formal (or at least, precise or rigorous) semantics for the UML diagrams, but these proposals have not been yet incorporated in the UML standard meta-model [4].

The OMG's UML 2.0 OCL RfP process is not finished, which implies that the OCL definition has not come yet with a final proposal for a precise object-oriented meta-modelling approach within the UML views [14]. Nevertheless, in their industrial projects, the authors are using UML with OCL 1.0 for dealing with non-functional requirements. Sequence diagrams with time inscriptions have been used for the specification of the canonical latency and duration constraints, which are viewed as composites for more accurate categories of timed requirements (for performance and safety constraints specification).

4.2.3 New layer in the project

Modelling the different system's views in UML and later transforming the multiple-view model into an implementation language imposes a new layer in the development process if compared with a situation where the systems are directly coded in the implementation language. This may be understood as a disadvantage because it implies that the designers must know one more language. A more optimistic perspective is however possible. If automatic code generation tools are available the final implementation language may be transparent to the designer, which means that he/she specifies the systems in UML, simulate their behaviour with the specifications, and pushes a button to obtain the system's implementation.

4.2.4 State Orientation

State models can be specified for the system's components that possess a complex or interesting dynamic behaviour. UML has two different meta-models for this purpose: statecharts and activity diagrams. Although these two meta-models present many important characteristics for reactive systems, namely concurrency and hierarchy, they do not allow an elegant treatment of the data path/plant resources management and the specification of dynamic parallelism. These are two crucial topics for complex, distributed and parallel embedded systems, since different parts of the system may require the simultaneous access to the same resource.

For embedded systems, the application of Petri nets (PNs) to the specification of the behavioural view is a proper alternative. PNs constitute a formal meta-model that can be simulated, formally analysed, and for which several implementation techniques are possible. In this context, for replacing UML's statecharts and activity diagrams, it is suggested the adoption of the shobi-PN, an extended object-oriented PN meta-model, to specify the reactive and dynamic behaviour of the system's software components, with the OCL 1.0 syntax to specify the non-functional requirements [11].

5. CONCLUSIONS

Based on the identified set of requirements for SLLs, this paper has discussed if UML can be used as an SLL for modelling the different views of embedded software systems. The answer to the question posed in the title is definitively positive. Some arguments were presented in what concerns the usage of UML as a solution to the problems faced by engineers when dealing with complex embedded software, namely in what concerns the user's requirements capture. For the system's requirements, UML lacks some adequate solutions, since the “statecharts+activity diagrams” approach is not satisfactory for describing the detailed behaviour in the presence of asynchronism, hierarchical level violations and dynamic concurrency.

A description of UML's main features for modelling embedded software is presented and its main advantages (standard, communication with the customer, object-oriented nature, platform independence, automatic code generation, extensions mechanism) and disadvantages (number of diagrams, not precise semantics, new project's layer, state orientation) are discussed within this field of software engineering.

6. REFERENCES

1. C. Ajluni. System-Level Languages Fight to Take Over as the Next Design Solution. *Electronic Design*, 48(2):68-78+110, Jan-2000.
2. G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
3. B.P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.
4. A. Evans, J.-M. Bruel, R. France, K. Lano, B. Rumpe. Making UML Precise. *13th Conf. on Object-Oriented Programming, Languages and Applications (OOPSLA'98)*, 1998.
5. J.M. Fernandes, R.J. Machado, H.D. Santos. Modeling Industrial Embedded Systems with UML. *8th Int. Workshop on Hardware/Software Codesign (CODES 2000)*, pp. 18-22, 2000. ACM Press.
6. J.M. Fernandes, R.J. Machado. System-Level Object-Orientation in the Specification and Validation of Embedded Systems. *14th Symp. on Integrated Circuits and System Design (SBCCI'01)*, 2001. IEEE CS Press.
7. R. Jigorea, S. Manolache, P. Eles, Z. Peng. Modeling of Real-Time Embedded Systems in an Object-Oriented Design Environment with UML. *3rd Int. Symp. on Object-Oriented Real-Time, Distributed Computing (ISORC 2000)*, pp. 210-213, 2000.
8. C. Kobryn. UML 2001: A Standardization Odyssey. *Communications of the ACM*, 42(10):29-37, Oct-1999.
9. E.A. Lee. What's Ahead for Embedded Software? *IEEE Computer*, 33(9):18-26, Sep-2000.
10. R.J. Machado, J.M. Fernandes, H.D. Santos. A Methodology for Complex Embedded Systems Design: Petri Nets within a UML Approach. *Architecture and Design of Distributed Embedded Systems*, B. Kleinjohann (ed.), chapter 1, pp. 1-10, 2001. Kluwer.
11. R.J. Machado, J.M. Fernandes. A Petri Net Meta-Model to Develop Software Components for Embedded Systems. *2nd Int. Conf. on Application of Concurrency to System Design (ICACSD'01)*, pp. 113-22, 2001, IEEE CS Press.
12. G. Martin, L. Lavagno, J. Louis-Guerin. Embedded UML: a merger of real-time UML and co-design. *9th Int. Symp. on Hardware/Software Codesign (CODES'01)*, pp. 23-28, 2001. ACM Press.
13. S. Narayan, D.D. Gajski. Features Supporting System-Level Specification in HDLs. *2nd European Design Automation Conference (EuroDAC - EuroVHDL'92)*, Sep-1992.
14. OMG. *Response to the UML 2.0 OCL RfP (ad/2000-09-03)*. OMG Document ad/2001-08-01, v. 1.0, Aug-2001.
15. F.J. Rammig. Approaching System-Level Design. *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, J. Mermet (Ed.), pp. 259-278, Kluwer, 1992.
16. W. Wolf. *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufmann, 2001.