Functional and Object-Oriented Views in Embedded Software Modeling

João M. Fernandes
Dep. Informática
Universidade do Minho
Braga, Portugal
jmf@di.uminho.pt

Johan Lilius
Computer Science Department
TUCS and Åbo Akademi University
Turku, Finland
Johan.Lilius@abo.fi

Abstract

The main aim of this article is to discuss how the functional and the object-oriented views can be inter-played in order to model the various modeling perspectives of an embedded system. We discuss if the object-oriented modeling paradigm, most likely the predominant one to develop nowadays software, in the broader sense of the term, is also adequate for modeling embedded software and how it must be conjugated with the functional paradigm. More specifically, we present how Data Flow Diagrams (DFDs), the main diagram in the traditional structured methods, can be integrated in an object-oriented development strategy based on the Unified Modeling Language (UML).

1 Introduction

In software engineering, when a new approach appears in the scene with the promise of solving all the problems faced by its professionals, the typical reaction is yet to abandon the old one. What actually happens is that ideas, concepts and techniques of both, the old and the new, approaches are merged and the final result is a combined solution. The object-oriented modeling paradigm is nowadays one of the most used approaches to develop software and, when it was proposed in the 80s, their advocates stated that it could overcome some, if not all, of the weaknesses associated with the structured methods. Some results indicate that, when the characteristics of the problem are well suited to an object-oriented approach, substantial time savings over traditional functional decomposition can be achieved in logical design [1]. But almost certainly we could make a similar claim in favor of structured methods if an adequate problem is used.

Setting up a framework for comparing analysis techniques and achieving useful conclusions is not an easy task. This may be the reason why there is not yet a definite "proof", even if not formal, that shows that the object-oriented paradigm is definitely better than structured methods [2], and some authors even suggest the reverse [3]. In fact, attempts to prove formally that one approach is better than another are seldom effective, in any domain. This is extremely harder in information technologies, because in real-world scenarios, there is hardly ever an opportunity to develop the same system in two different and independent ways and compare them.

If a careful comparison is undertaken, one can see that object-oriented and structured methods do not differ so much on the meta-models they use. For example, the set of diagrams suggested by the OMT methodology is, according to M. Jackson, *surprisingly* close to the traditional proposals of Structured Analysis [4]. In our opinion, there is not too much surprise in this fact, since object-orientation can, in an historical perspective, be seen as an evolution (and not a revolution) of the structured methods. Some authors even assume a more drastic position, by considering that "*object-oriented methods are structured methods, just like all the others that precede them*" [5].

In fact, object-oriented and structured methods both recognize the need to use three models to specify a complex software system: a functional model, a control model and a data model. For example, the usage of statecharts was proposed in both approaches apparently with successful results [6]. Additionally, the now classical software engineering techniques and guidelines, originally conceived for structured design, namely modularity, data hiding, low module coupling, and high module cohesion, are still relevant and useful in object-oriented design [7].

The major discrepancy between structured and object-oriented analysis relies presumably on the way those three models are used, namely, the order in which they are created. Object-oriented methods have the class diagram (a data-oriented model) as its main modeling tool, while struc-

Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)
0-7695-2125-8/04 $ 20.00 © 2004 IEEE

tured methods use DFDs (an activity-oriented model) as its principal diagram. The popularity of object-orientation is probably due to the observable emphasis on data in system design that has increased considerably in the last years. It is our belief however that in embedded systems an activity oriented view of the system is typically more useful.

In this paper, we explore in detail how to integrate DFDs into UML. This integration could look superfluous, since UML is a huge language with many modeling elements, that is considered adequate and useful for a great number of application areas. However, it is not at all an universal language that could be deployed in any problem domain. Specifically, UML does not include DFDs or any similar diagram, which represent, in our opinion, a useful model for some kinds of software, namely embedded software.

The proposed integration could also look forced or antinatural, because we are trying to unite two apparently discordant approaches for developing software systems. Nonetheless, in our opinion, software engineers should not take a religious or dogmatic attitude when it comes to choose or use a specific model. We believe that currently the question that must be answered by the software engineers is not which models to create, but how to nicely integrate different models, if all of them are deemed valuable for the description of the system. This question is, in fact, a today's problem, when UML, for example, is adopted as a modeling language, because it includes several diagrams that are only loosely related. The proper integration of theories and concepts is considered nowadays as one of the key challenges in the field of embedded systems:

> "*Our answer to the question of what are the new theoretical challenges raised by the (. . .) field of embedded systems is that, what we need, is not a new theory of embedded systems. (. . .) What is required is the integration of the relevant theories and methods into a coherent development process and making it work.*" [8].

This paper discusses the unification of two different modeling perspectives: the functional and the object-oriented views. The discussion is especially oriented towards the development of embedded software, but we believe that the ideas and arguments presented here can also be adapted in a large extent to other types of software. In addition, we also focus the attention on the analysis phase of the development process, giving less importance to the other phases, namely design, implementation, and test.

This paper is based on a Technical Report [9], where a full list of references can be found. Additionally, the proposals made here to unify the functional and the object-oriented views are used in the technical report to model an IPv6 router system. It is also important to notice that the UML version under consideration in this paper is 1.4.

2 Functional and object-oriented views

We understand the functional view, also designated dynamic or behavioral, as the system's perspective that centers around the behavior of the system. Similarly, the object-oriented view is understood as the perspective that focus on the structure of the system, namely its data. In fact, it is commonly acknowledged that one major component of the object-oriented analysis techniques is based on the Entity-Relationship concepts [10].

For complex systems, it is inevitable that structural and dynamic models have to be intertwined or interplayed, during the development activities, at different moments and also at distinct levels of abstraction. For instance, the whole system can be seen as a module and a state-machine can be devised for it. We can later decompose the system in sub-systems and create, for each one, an activity diagram that represents the respective function. The sub-systems can, by themselves, be decomposed in objects, which can have their life-cycle represented by a Petri net. We can go as many levels as we want and, as modelers, we are always changing from structural models to dynamic ones and vice-versa. The same combination appears to occur, at an orthogonal perspective, with specification and implementation [11].

A similar systemic view was proposed in [12]. There, a combination of Finite-State Machines (FSMs) with other concurrent models of computation (namely, dataflow, synchronous/reactive and discrete event) is suggested. The idea is that an FSM can be nested within a module in a concurrency model, which is to be interpreted as the FSM describing the behavior of that module. Conversely, a subsystem in some concurrency model can be nested within a state of an FSM, which means that the subsystem is active only when the FSM is in that specific state. The hierarchy can be placed anywhere and is arbitrarily deep. A proposal with identical practical consequences is the "tool box" approach to software specification, where each system's module may be specified individually using the technique most adequate for it [13]. This approach seems very useful for specifying complex systems, that are generally composed of several components, each one with its own idiosyncrasies.

One of the main strengths of these approaches is that, for example, the concurrency model can be selected to best suit the problem at hand, based upon its particular characteristics. Consequently, developers are not restricted to a single meta-model, as usually occurs. Hence, the following meta-models, which seem useful for embedded computing can be adopted and mixed: continuous time and differential equations, discrete time and difference equations, state machines, synchronous/reactive models, discrete-event models, cycle-driven models, rate monotonic scheduling, synchronous message passing, asynchronous message passing, timed CSP, publish and subscribe [14].

Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)
0-7695-2125-8/04 $ 20.00 © 2004 IEEE

Unfortunately, it exists a culture of rivalry in the software community with respect to the two major paradigms. Nowadays, the convention is to use either a "pure" object-oriented approach or a "pure" functional approach. We prefer to view them as complementary, each one with its own strengths and weaknesses. We think that a proper mixture of the approaches is possible, so that the best of both worlds can be achieved. There were several attempts to combine these two approaches [15] [16] [17] [18], but none of them is widely used. Although some researchers [19] argue that object-oriented analysis and structured analysis are fundamentally incompatible, we believe that the topic deserves more research effort in order to understand if the integration can be effectively achieved and, if a positive answer is obtained, how that can be accomplished.

In fact, merging divergent aspects or ideas appears to be a recurring solution in many areas of knowledge, with extremely good results in some cases. Werner K. Heisenberg, 1932 Nobel Prize laureate in Physics, observed that:

> "*It is probably quite true generally that in the history of human thinking the most fruitful developments frequently take place at those points where two different lines of thought meet. These lines may have their roots in quite different parts of human culture, in different times or different cultural environments or different religious traditions: hence if they actually meet, that is, if they are at least so much related to each other that a real interaction can take place, then one may hope that new and interesting developments may follow.*" [20].

Computing science seems also to benefit when opposite or dualistic aspects are taken into consideration. Indeed, significant improvement had always been achieved when the fruitful integration of a dual pair was possible [21]. That observation was also a motivation for this work.

3 UML

One common aspect of structured and object-oriented methods is that they usually adopt graphical notations for describing the system under analysis. For a graphical notation to be useful it must be clear and intuitive, so that both clients and designers can understand it, but also precise and rigorous, so that computer tools can analyze, simulate and validate it. One drawback of graphical representations is that they are not adequate for capturing detail. A graphical model that has excessive information becomes as hard to read as an equivalent textual description.

One of the languages that is gaining exponential popularity and usage is the Unified Modeling Language (UML). UML is a graphical modeling language, that supposedly unifies and integrates the different notations used before by several software methodologies. This notation became a real necessity, because, between 1989 and 1994, the number of object-oriented methods increased from fewer than 10 to more than 50 [22]. UML constitutes the *de facto* standard notation and semantics for properly describing software built with object-oriented or component-based technology. It is undoubtedly a step in the right direction, but it is not a perfect or universal modeling language [23]. We believe that UML, as it stands today, must be, in some contexts and for some application domains, complemented with other meta-models or at least adapted (by stereotyping it) to address those meta-models.

The main problem with UML is that its semantics is not precise. This is a recurring problem of graphical notations, since they seem to carry a higher risk of vagueness than textual languages; for instance, lines and boxes suggest less need of preciseness than identifiers and assignment statements [4]. The reason for this to happen is that graphical languages are typically used without a compiler. Although this can also occur with textual languages (we can use a simple text editor to write the program), it is more frequent to process them with compilers. Therefore, the exclusive use of graphically-based and intuitive notations is often insufficient for correctly specifying a given system.

Since UML is a multiple-view meta-model, a serious consequence is that inconsistencies, among the diagrams used for specifying the system, may occur. This also happens when the designers are using computer tools for editing the diagrams, since usually those tools do not perform all kinds of checks necessary to guarantee full consistency. In large projects, where it is common to have several team members modifying the same set of diagrams, consistency is even more difficult to guarantee. Several attempts have been made to remedy this problem, because it usually proves to be costly in software development projects.

The most common way to use UML diagrams during analysis is to start with use case diagrams and to proceed with sequence diagrams, to describe some scenarios of the interaction between the system and its actors. Later, a class diagram is created, taken into consideration the previous diagrams. Usually a state-chart diagram is associated to each class for describing the corresponding behavior.

Although UML includes nine diagrams, using only the referred four during analysis seems to be sufficient for the majority of developers. In fact, collaboration diagrams are not included, because they are similar to sequence diagrams, activity diagrams are usually ignored, since they represent a subset of statechart diagrams and component and deployment are not at all used or only used in later development stages.

We find two major problems with this typical usage, in what concerns the development of an embedded system.

Firstly, the "jump" from use cases and scenarios to classes is, in our opinion, a very big one. This step requires too much ingenuity and there is not an evident direct relationship between use cases and classes. We think that there exist many similarities between this transformation step and the transition from analysis to design in structured methods, which was vastly criticized to be one of the biggest limitations of those methods. Instead, what we need to develop complex embedded systems is a seamless process, from requirements until the coding phase, that preserves the behavior and integrity of the models in each development step [8].

Secondly, for embedded software, the attention should be focused towards object diagrams, instead of class diagrams. The majority of the methodologies for developing software do not pay too much attention to the object diagram. In fact, software developers concentrate too much on the class structure and too little on the object structure [24].

Finally, it is important to discuss what are the most typical mistakes that prevent organizations to get more value from using UML in their software projects [25]. Firstly, it is crucial that the UML models do not possess a level of detail similar to the final executing system. Models are abstractions of the reality and serve "only" to visualize, specify, document the software. Therefore, producing quality executable programs is, almost always, the main aim of any project. Secondly, some companies utilize UML just as a documentation notation. This is a very limited way of taking advantage from UML, since using it as a communication medium among the various stakeholders proves usually useful. Thirdly, it is important that every developer should use and understand UML to get the most value from it. If programmers (i.e., the developers that actually write the final code) do not strongly rely on the UML models to construct the system, then big mismatches between the UML models (that represent the user's and system's requirements) and the final system will arise easily and naturally.

4 Combining DFDs with UML

The combined usage of DFDs with other UML models can be accomplished in several ways and this combination must be interpreted in a very broad sense. This results from the fact that the development of a software system proceeds in steps, where several different models are being refined and detailed, but also transformed, merged, split, integrated, etc. Therefore, in this context, the term "combined" used above can mean several different things. One possibility is that DFDs are used during the development process and that they are transformed into UML diagrams or vice versa. A distinct interpretation consists in not using DFDs at all, and give some UML diagram a DFD flavor. Another possible alternative is to use DFDs and UML diagrams, and propose techniques for integrating their usage.

Under these circumstances, the question that is important to answer is how and when can DFDs be used within the development process of an embedded system. The way to tackle this question can be divided in three more specific ones, to which we hope to give real answers in this paper:

1. Are DFDs an useful model for embedded software?
2. In which phase of the development process must DFDs be introduced?
3. Which views should DFDs cover?

We do think that DFDs can be, in some contexts, an useful model for embedded software. This idea can be largely confirmed by the widespread usage of data-flow oriented meta-models for describing digital and embedded systems, namely Process Networks [26] and Control/Data Flow Graphs (CDFG) [27]. It is important to stress that the meta-model behind CDFG has many resemblances to the one associated to DFDs with control extensions (as proposed in the Ward-Mellor method, for example).

However, for developing embedded software, we do not believe that it is possible to rely on a "one-size-fits-all" solution, due to the wide range of applications covered by this software field. This means that in some situations DFDs may be an adequate model of computation, but that in others they are not. We think that a data-flow model may be the most adequate one for transformational systems, that is, systems that continuously repeat the same data transformation on streams of data [28]. Application areas, where the data-flow paradigm of computation is evidently useful and widely adopted, include for example multimedia systems, telecommunication devices, and digital signal-processing systems. Furthermore, DFDs are also very good at producing systems based on a menu structure, because the idea of functional decomposition and leveling is just right for a menu-based development.

The incorporation of DFDs into UML can not be made without first deciding if they are merely added as a new diagram or whether it is possible to view them as an extension or adaptation of an existing UML diagram. The combined use of DFDs with other UML models, if deemed useful, can be accomplished with at least two approaches (fig. 1). In the first alternative, the DFD meta-model is mapped into UML concepts, while in the second both meta-models are available as originally devised.

We think that the first alternative is preferable, because it allows us to restrict to the UML meta-model in what concerns the model's back-end processing (model transformation, validation, code generation). This restriction allows the usage, without any modification, of any tool that supports UML for edition, documentation, validation, simulation and code generation purposes. However, this solution forces the DFDs to be adapted to a given UML diagram,

which means that we are not able to use DFDs at their maximum expressiveness. Another argument in favor of the first alternative is that almost all people involved in UML agree that it already offers a reasonable number of modeling diagrams, sufficient for the vast majority of modeling purposes, and that it should not be further extended, namely in what concerns the number of diagrams.

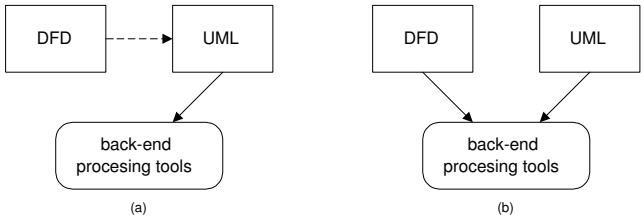

Figure 1. Two alternatives for integrating DFDs in UML: (a) DFDs mapped into UML concepts; (b) DFDs added to the UML meta-model.

In any case, to take full advantages of DFDs, the designer must be completely aware of their associated meta-model. So, even in the situation where a UML diagram is adapted to be viewed as a DFD, the designers have to understand the complete set UML+DFD. Under this assumption, the argument that adapting a UML diagram may result in confusion and misunderstandings is not a strong one.

We propose three major ways of using, in an integrated way, DFDs within an object-oriented system development.

1. DFDs to refine the use case model;
2. DFDs to detail the behavior of a system's component;
3. DFDs to be transformed into class diagrams.

The rationale behind these proposals is always to have, as the major model to drive the implementation phase, some object or class diagram, so that an object-oriented programming languages can be used, but also to include the DFDs in the modeling process.

A full discussion of these topics with a more complete list of references can be found in [9]. Due to space limitations, we will concentrate on the first proposal, and only shortly discuss points 2 and 3.

5 DFDs to refine the use case model

It is commonly accepted, within the object-oriented community, that the analysis of a software system should be started with uses cases. A use case diagram represents a functional view of the system. Similarly, in structured methods, a system is seen as a provider of functions to the user, which is an adequate view for requirements capture.

However, using use cases does not necessarily imply that subsequently an object-oriented approach must be followed. Use cases represent a technique that is quite independent of object-oriented modeling and can be applied to any system, developed either with a structured or object-oriented approach [29]. In any case, adopting use case diagrams should not be seen as an opportunity to follow again a functional decomposition of the system. This is the reason for our proposals to incorporate always object-oriented diagrams in the modeling process.

In this context, it seems that the transformation of a use case model into a DFD-like model is not at all awkward or forced, since both meta-models can be used naturally for focusing on the same modeling perspective. DFDs can be made more detailed, since they include processes (similar concept to use cases) and external entities (identical to actors in use case models), but also data stores and data flows, which indicate data-dependencies among processes and are not directly representable in a use case diagram. Even though UML provides two relationships, «include» and «extend», to connect use cases among them, they are not related to data or control flows, but rather with dependencies between use cases. We will not explore further this topic, since the relationships seem to confuse the designers, instead of helping them.

We clearly notice that DFDs are more detailed than use cases. As a matter of fact, it is usually difficult to perceive how use cases interact, especially whenever there are many of them in a diagram. An interesting solution to this limitation is to use an activity diagram that shows how use cases are related and also alternatives and decisions [30].

As already stated, we would like, if possible, to use UML as the notation to represent the systems being modeled. Therefore, the meta-model behind DFDs must be mapped into UML concepts. Generically speaking, any UML diagram could be used for this purpose, as long as stereotypes are associated to its constructs. In the extreme case, we were only using the syntax of the diagrams, but would associate a very different semantics to it. But we prefer to adapt a UML diagram whose respective model of computation is as close as possible to the DFD's one. However, this choice should be taken with care, since different diagrammatic representations do not necessarily have the same effectiveness or computational power [31].

Before choosing which UML diagrams best match with DFDs, it is important to notice that DFDs are not representing only the behavior of the system. We can also think about DFDs as defining a given structure or architecture for the application being analyzed: they are dividing or decomposing it in its modules or subsystems and also showing the communication paths amongst those modules. As a mat-

ter of fact, DFDs can be used to describe only the structure of a system, showing just its components and the channels through which information flows [32]. With this view on DFDs, no behavioral aspect is being modeled.

In the case of DFDs, the behavior is usually organized as a tree of processes and only the leaf ones (also called *functional primitives*) must be associated with a description, traditionally a PSPEC (Process Specification), that specifies concisely and briefly the intended behavior. In fact, when a system is divided in parts both structure and behavior are being decomposed. For example, some authors consider that the design methods that have evolved to work with object-oriented, procedural and functional languages all tend to break the systems down into units of behavior [33]. They consider that all systems are submitted to a functional decomposition, even if, for each computational paradigm, different units of behavior (objects, procedures, and functions) are considered. Similarly, it seems thus acceptable that we consider that all those design methods force equally the systems to a structural decomposition.

5.1 Selection of a UML diagram to represent DFDs

Considering that DFDs can be viewed as a structural notation, we had considered initially the following UML diagrams as possible solutions to represent DFDs: object diagrams, collaboration diagrams, component diagrams, deployment diagrams, and activity diagrams. Despite the fact that class diagrams constitute also a static model, they are not considered here as a primary option, because we consider that for developing an embedded system an object diagram is more valuable than a class diagram [34].

Component diagrams and deployment diagrams were rejected from the very beginning, because they are intended to represent the physical layout of a software system. Additionally, these so-called implementation diagrams are still in a very elementary form. Object diagrams were also not considered as an alternative, since a collaboration diagram with no messages is equivalent to an object diagram. The meta-model of collaboration diagrams is indeed a superset of that for object diagrams.

Activity diagrams could also look as the best candidate for this adaptation. At first glance most people think that activity diagrams look like DFDs. Although both diagrams are activity-oriented models, there are some fundamental distinctions between them. Not realizing these distinctions and concentrating just on the similarities appears to be one of the main difficulties for practitioners to use activity diagrams, because it is not easy to make the shift from data-oriented to functional-oriented thinking. The problem is that activity diagrams show control dependencies among activities, rather than data ones as happens with DFDs. In addition, an activity diagram constitutes a state diagram that models a sequence of actions and conditions taken within a process, while a pure DFD can model concurrent processes without considering control decisions.

Our opinion is that **collaboration diagrams** constitute the most appropriate UML model for representing DFDs. The decomposition that DFDs impose could be equally achieved with collaboration diagrams. Although collaboration diagrams and DFDs could look similar, at least superficially, there is however an important difference. DFDs constitute a static view of the system, in the sense that all the system's connections and all its processes, used during the system's life cycle, are represented. Contrarily, a collaboration diagram represents a dynamic view of the system and allows the visualization of a unique point in time, showing what are the interactions within a particular subset of the objects that a system is composed of. This means that collaborations diagrams can be adapted but also that they must be slightly modified.

For example, the ROOM methodology [35] and its UML-based successor, UML — Real Time [36], propose an approach based on collaboration diagrams. The actor concept of ROOM is captured by the UML-RT's «capsule» stereotype, which is a specialization of the class concept. Simple capsules have their functionality realized directly by the associated state machine, while complex capsules combine the state-machine with a network of internal subcapsules. This internal architecture is specified as a collaboration diagram.

In the MOOSE methodology, the similarity between the *Object Interaction Diagrams* (OIDs), that show the interactions among objects, and the structured analysis' DFDs is also noted [37]. The main difference lies on the semantics for the objects interactions, which is strongly distinct from the semantics for the passage of data between processes.

Embedded system specification and design consists in the description of the system's desired functionality and in the mapping of that functionality for implementation by a set of components [28]. Thus, starting from use cases, to describe the system functionality, and proceed to objects, to specify the components of the system, as proposed here, addresses directly those aspects. Even if some compromises are to be considered, the main idea is that collaboration diagrams can be viewed as representing simultaneously the architecture of the system and its data-flow view. If this approach is taken into account, DFDs are to be seen as a refinement of use cases, and so they can represent the whole system. Although this seems to contradict the recommendation that DFDs should not be used as the main diagram to represent the whole system, we believe that this is not the case. In fact viewing collaboration diagrams as DFDs, does not imposes a functional decomposition of the system, since the DFDs' processes are now represented as objects.

Thus, we can view the system according to its data-flow view, even if it is essentially an object-oriented or object-based system.

5.2 Transforming use cases into objects

If use case diagrams are to be transformed into DFDs, represented as collaboration diagrams, the main question is thus how to transform use cases into objects, since these are the constituents of collaboration diagrams. This kind of transformation is not simple and easy at all and face several problems. Firstly, despite the existence of some proposals for automatically obtaining objects, namely the SysObj tool [38], it generically involves several decisions that can not be done by a method or a tool, caused by the natural discontinuity between functional and structural models.

Holland and Lieberherr go a little further and consider that the identification of objects and the description of the relationships between them are two of the three challenges of object-oriented design [7]. In fact, the rules for a given domain are defined by the relationships among things and their formalization as associations of various kinds are often far more interesting than the objects [30].

To tackle these crucial questions, namely the identification of objects from use cases, some proposals exist [39] [40], but usually they concentrate on classes rather than real objects. This difference, that might apparently look superficial, entails a distinct approach and focus. A strategy, called *4-Step Rule Set* (4SRS), was already devised to assist the designers in the transformation of use cases into objects [41].

The 4SRS associates, to each object found during the analysis phase, a given category: interface, data, control[1]. Each one of these categories is intimately related to one of the three orthogonal dimensions, in which the analysis space can be divided (information, behavior and presentation) [39]. This categorization gives rise to object models that, in their essence, are similar to the architectures imposed by the *Model-View-Controller* (MVC) pattern [42]. The division has also strong resemblances to the typical 3-tier client/server architectures commonly used within *Enterprise Resource Planning* (ERP) systems, which divide the software application into three layers: the presentation, the business logic, and the database.

An interface-object models behavior and information that depend on the system's interface, i.e., the dialogue of the system with the actors that interact with it. A data-object predominantly models information, whose existence must be lengthy (temporary storage should not be modeled as data-objects). Apart from the attributes that characterize the data-object, the behavior associated to the manipulation of that information must also be included in the data-object.

[1]It is also possible to designate these three categories as boundary, entity, and function, respectively.

A control-object models behavior that can not be naturally associated to any other object. For instance, the functionality that operates on several objects and that returns a result to an interface-object is a control-object.

With this categorization of objects, object and collaboration diagrams become similar to DFDs that are composed of data stores, processes, and external entities. We think that it is relatively easy to adapt the main ideas of the 4SRS to transform use cases diagrams into DFD-like diagrams, and that this transformation is valuable to develop embedded software. However, it is crucial to avoid creating excessive functional control-objects that dictate the behavior of data-objects, with no associated "intelligence". In fact, there appears to be a strong tendency, which is important to contrariate, for control-objects to usurp the responsibilities of data-objects [43]. Furthermore, it is not unusual to see data-objects and control-objects becoming respectively the data representation and the processes, i.e., to have a clear separation between data and processes that object-orientation was supposed to avoid. Thus, we emphasize that an object, independent of its category, should be viewed as a rich modeling entity with both attributes and methods, and, eventually, a state-oriented model associated with it.

This approach leads naturally to a component-based modeling style, because the objects can be seen as logical components, which hide their internal details and accomplish the communication to other components through well-defined interfaces. The objects that are created by the 4SRS must be viewed at a higher level of abstraction if compared with the traditional perspective in object-oriented analysis and design. The objects are not to be viewed as, for example, a stack or a queue, which have a small scope, are centered on data and are passive. When developing complex systems, some lower-level classes will be used for sure, but generally these classes are not visible during analysis or even design. We must see an object as a component of the system. This view is similar to ROOM's one, where they define "*an object as a software machine, or as an active agent implemented in software*" [35]. In ROOM, a wider perspective is even taken and an object is additionally defined as "*a logical machine, which is an active component of a system and which may be implemented as software, as digital hardware, or even with some nonelectronics-based technology*".

In fact, within the 4SRS, data-objects can be seen as data stores. The data store notation in DFDs is used to save information that is used within the system. Although data-objects are much richer than data stores, since they can also have associated methods, this perspective does not conflict with the object-oriented view of data objects.

The data being modeled can be as small as an item (variable or record) or as big as a table or even a complete database. However, it is more adequate to view the DFDs'

data stores at a very high level of abstraction. In other words, DFDs should not be used to model the details of the information perspective of the system, since other diagrams are used for that. One proposal that follows this view suggests an adaptation of DFDs, where each data store symbol is thought to represent a complete database rather than a single table [44]. This avoids redundancies and conflicts with the data model of the system, usually represented by an entity-relationship diagram or a class diagram.

The interface-objects can be equally understood as ports of the system. For every actor connected to a use case[2], it is necessary to introduce an interface-object to handle the communication between the actor and the system. Alternatively, interface-objects can be seen as the processes responsible for receiving the inputs and/or sending the outputs, when that perspective makes sense.

The control-objects can be viewed as DFDs' processes. They are used to operate on data received from the outside (from an interface-object) or stored internally (in data-objects) and to generate new data to be sent to the outside (to an interface-object) or stored internally (in data-objects).

6 Other uses of DFDs

6.1 DFDs to detail the behavior of a component

We do not explore in detail this hypothesis of using DFDs in this paper. This possibility was already suggested, for example, by Ivar Jacobson in a conference panel [19]. Briefly, we can comment that the UML meta-model defines an association between *ModelElement* and *Statemachine*, called *behavior* [45, p. 2-145]. Almost all the elements that can be included in the UML diagrams are ModelElement. However, there is also the following well-formed rule [45, p. 2-156]:

```
self.context.notEmpty implies
  (self.context.oclIsKindOf(BehavioralFeature) or
   self.context.oclIsKindOf(Classifier))
```

This means that only behavioral features and classifiers can have state machines. A BehavioralFeature is a method of a class and a Classifier can be a Class, a Use Case, an Actor, a DataType, a Component, an Artifact, a ClassifierRole, an Interface, a Subsystem and a Signal.

This means that we can define the behavior of any classifier element using a statechart (or an activity diagram). Thus, it is possible to update this association so that we can define the behavior of a model element using a statechart, an activity diagram but also a DFD diagram.

With this approach, it is fundamental to realize that DFDs are not being adopted as the main description for specifying the systems. If we follow this guideline, the problems of top-down functional decomposition are avoided, but the benefits of their data-flow flavor still remain. In UML this aim can be easily achieved since it promotes a multiple-view modeling approach, thus distributing the different system's views to several diagrams.

The main disadvantage of this approach is that it forces the designer to use DFDs as they are, and thus forces the back-end tools to support both DFDs and UML.

6.2 DFDs to be transformed into object/class diagrams

Assuming that generically DFDs are not an adequate tool for capturing the user's requirements, they are however useful in later phases of the development. One specific situation where the usage of DFDs is helpful is in re-engineering activities if the system was previously developed following the guidelines of some structured method. Even if the diagrams are no longer available, it is expected to be easier to reverse-engineering the program code into DFDs and other complementary models, than to transform it directly into some object-oriented models.

Therefore, we propose that DFDs could be transformed into object or class diagrams. Some similar ideas were already proposed in the FOOM methodology [18] for developing information systems, but its usage for embedded systems requires necessarily some adaptation. The transformation of a functional specification in Z into an object-oriented one in Object-Z, for re-engineering purposes, is also proposed in [17].

7 Conclusions

Although the combination between the functional and the object-oriented approaches is almost universally seen as a "bad" approach to software modeling, we believe that it can give, in some specific situations, good results, if not seen as an infallible solution, but instead used with some precaution. We believe that for programming purposes (i.e., for the process of creating a text-based program from the models that describe the system's behavior and architecture), object-oriented programming languages offer many advantages that should not be put apart by any organization that develops software, embedded one included.

In this paper the combination of the functional and object-oriented approaches, represented respectively by DFDs and UML was analyzed. The emphasis of the discussion was put in the questions related to the analysis phase and to embedded software systems. The rationale was always to have, as the major model to the implementation phase, some object or class diagram, so that an object-oriented programming languages could be used, but also to

[2] In a use case diagram, it is possible to have actors that are not connected to use cases. An actor of that type is called secondary.

include DFDs in the modeling process. We have suggested three main directions to achieve that combination: (1) DFDs to refine the use case model; (2) DFDs to detail the behavior of a system's component; and (3) DFDs to be transformed into class diagrams, in a re-engineering situation.

In fact, it is quite intriguing why the usage of use cases, within the context of object-oriented development, is so popular and considered a suitable technique, if they, similarly to DFDs, decompose functionally a system. The answer, in our opinion, lies on the fact that use cases are a simple technique to understand and use, and produce good results in several situations.

For some types of embedded systems, where the system is constructed to obey a specific standard, and not to fulfill the needs and expectations of human users, the usage of DFDs is, for modeling purposes, more adequate than use cases diagrams. Use case modeling is quite useful when the development team needs to discuss the requirements of a system with its stakeholders, especially the users, managers, customers, and clients. This occurs because use case diagrams are an easy-to-read notation and, due to their extremely simplicity and the intuition behind the concepts of use case and actor, promote the participation of the non-technical stakeholders. This characteristics is not so important for some types of systems, such as digital-signal processing systems, that do not have human users or that are data-triggered and whose functionalities are to be executed in a particular sequence. In contrast, DFDs are good for systems that present these characteristics.

Taken in consideration that DFDs are more expressive than use case diagrams, they could be used as use case diagrams, for users' requirements capture, omitting thus some of their constructs (for example, data stores). Later, more detailed information could be added, by the designers, this time without the user's intervention. Based on the DFDs produced, obtaining an object-oriented architecture should be possible (although we do not claim that it is easy or simple). If this is agreed to be suitable from use case diagrams (in conjunction with other models, such as sequence and collaboration diagrams), that should also be possible, and easier we ought to add, with DFDs and those same additional diagrams.

As future work, the following topics deserve more attention. Firstly applying the techniques proposed here to complex examples would allow more solid assessments about the usefulness of those techniques to be drawn. Secondly, a solid integration of DFDs with UML can not be only based in using both in a combined way at the process-level. Additionally, it is fundamental to investigate, at the semantic and meta-model levels, what are the implications and consequences of that combination. Thirdly, analyzing the effective ways of extending the 4SRS is also a future path for continuing this work.

Acknowledgments

The first author acknowledges the financial support from CIMO, under grant HH-02-383, that partially supported his post-doctoral studies at TUCS, and from FCT and FEDER under project METHODES (POSI/37334/CHS/2001).

References

[1] J. Kim and F. J. Lerch. Towards a Model of Cognitive Process in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies. In *Conference on Human Factors in Computing Systems (CHI '92)*, pp. 489–98. ACM Press, May 1992.

[2] R. L. Glass. The Naturalness of Object Orientation: Beating a Dead Horse? *IEEE Software*, 19(3):103–4, 2002.

[3] I. Vessey and S. A. Conger. Requirements Specification: Learning Object, Process, and Data Methodologies. *Communications of the ACM*, 37(5):102–13, 1994.

[4] M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press, 1995.

[5] D. J. Hatley, P. Hruschka, and I. A. Pirbhai. *Process for System Architecture and Requirements Engineering*. Dorset House, New York, 2000.

[6] B. P. Douglass, D. Harel, and M. Trakhtenbrot. Statecharts in Use: Structured Analysis and Object-Orientation. In G. Rozenberg and F. Vaandrager, editors, *Lectures on Embedded Systems*, vol. 1494 of LNCS, pp. 368–94. Springer-Verlag, 1998.

[7] I. M. Holland and K. J. Lieberherr. Object-Oriented Design. *ACM Computing Surveys*, 28(1):273–5, 1996.

[8] A. Pnueli. Embedded Systems: Challenges in Specification and Verification. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software, Second International Conference, EMSOFT 2002*, vol. 2491 of *Lecture Notes in Computer Science*, pp. 1–14. Springer-Verlag, Oct. 2002.

[9] J. M. Fernandes. Functional and Object-Oriented Modeling of Embedded Software. Technical Report 512, TUCS, Turku, Finland, Feb. 2003.

[10] P. P.-S. Chen. *Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned*. In M. Broy and E. Denert, editors, *Software Pioneers: Contributions to Software Engineering*, pp. 297–310, Springer-Verlag, 2002.

[11] W. Swartout and R. Balzer. On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–40, 1982.

[12] A. Girault, B. Lee, and E. A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–60, June 1999.

[13] W. G. Howerton and M. G. Hinchey. Using the Right Tool for the Job. In *6th IEEE International Conference on Complex Computer Systems (ICECCS '00)*, pp. 105–15. IEEE CS Press, Sep. 2000.

[14] E. A. Lee. Computing for Embedded Systems. In *18th IEEE Instrumentation and Measurement Technology Conference (IMTC/2001)*, May 2001.

[15] B. Alabiso. Transformation of Data Flow Analysis Models to Object Oriented Design. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pp. 335–53. ACM Press, 1988.

[16] P. T. Ward. How to Integrate Object Orientation with Structured Analysis and Design. *IEEE Software*, 6(2):74–82, 1989.

[17] K. Periyasamy and C. Mathew. Mapping a Functional Specification to an Object-Oriented Specification in Software Reengineering. In *24th ACM Annual Conference on Computer Science (CSC '96)*, pp. 24–33. ACM Press, 1996.

[18] P. Shoval and J. Kabeli. FOOM: Functional- and Object-Oriented Analysis & Design of Information Systems — An Integrated Methodology. *Journal of Database Management*, 12(1):15–25, 2001.

[19] D. de Champeaux, L. Constantine, I. Jacobson, S. Mellor, P. Ward, and E. Yourdon. Panel: Structured Analysis and Object Oriented Analysis. In *European Conference on Object-Oriented Programming / Conference on Object-Oriented Programming Systems, Languages and Applications (ECOOP/OOPSLA '90)*, pp. 135–9. ACM Press, Oct. 1990.

[20] W. Heisenberg. *Physics and Philosophy: The Revolution in Modern Science*. Harper & Row, 1958.

[21] A. C. Sodan. Yin and Yang in Computer Science. *Communications of the ACM*, 41(4):103–111, 1998.

[22] G. Booch. UML in Action. *Communications of the ACM*, 42(10):26–28, 1999.

[23] G. Engels, R. Heckel, and S. Sauer. UML — A Universal Modeling Language? In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000*, vol. 1825 of LNCS, pp. 24–38. Springer-Verlag, June 2000.

[24] S. Sigfried. *Understanding Object-Oriented Software Engineering*. IEEE Press, 1996.

[25] G. Booch. Growing the UML. *Software and Systems Modeling*, 1(2):5–9, 2002.

[26] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[27] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufman, Sep. 2000.

[28] D. D. Gajski and F. Vahid. Specification and Design of Embedded Hardware-Software Systems. *IEEE Design & Test of Computers*, 12(1):53–67, 1995.

[29] I. Jacobson. Basic Use Case Modeling (Continued). *Report on Object Analysis and Design*, 1(3):7–9, 1994.

[30] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Object Technology. Addison-Wesley, 2002.

[31] J. Hahn and J. Kim. Why Are Some Representations (Sometimes) More Effective? In *20th International Conference on Information Systems (ICIS '99)*, pp. 245–59. Association for Information Systems, 1999.

[32] D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff - Part I: The Basic Stuff. Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, Sep. 2000.

[33] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97 - Object-Oriented Programming, 11th European Conference*, vol. 1241 of *Lecture Notes in Computer Science*, pp. 140–9. Springer-Verlag, 1997.

[34] J. M. Fernandes, R. J. Machado, and H. D. Santos. Modeling Industrial Embedded Systems with UML. In *8th Int. Workshop on Hardware/Software Codesign (CODES 2000)*, pp. 18–22. ACM Press, May 2000.

[35] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[36] B. Selic. Using UML for Modeling Complex Real-Time Systems. In F. Mueller and A. Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98*, vol. 1474 of LNCS, pp. 31–40. Springer-Verlag, June 1998.

[37] D. Morris, G. Evans, P. Green, and C. Theaker. *Object-Oriented Computer Systems Engineering*. Applied Computing. Springer-Verlag, London, UK, 1996.

[38] L. B. Becker, C. E. Pereira, O. P. Dias, I. M. Teixeira, and J. P. Teixeira. MOSYS: A Methodology for Automatic Object Identification from System Specification. In *3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pp. 198–201. IEEE CS Press, Mar. 2000.

[39] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[40] D. Rosenberg and K. Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*. Object Technology. Addison-Wesley, 1999.

[41] J. M. Fernandes and R. J. Machado. From Use Cases to Objects: An Industrial Information Systems Case Study Analysis. In *7th International Conference on Object-Oriented Information Systems (OOIS '01)*, pp. 319–28. Springer-Verlag, Aug. 2001.

[42] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[43] R. Pawson. Naked Objects. *IEEE Software*, 19(4):81–3, 2002.

[44] I. Millet. Technical Note – A Proposal to Simplify Data Flow Diagrams. *IBM Systems Journal*, 38(1):118–21, 1999.

[45] OMG Unified Modeling Language Specification. Technical report, OMG, Sep. 2002.