CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Universidade do Minho: RepositoriUM

Softw Syst Model (2006) 5:403–428
DOI 10.1007/s10270-006-0013-0

REGULAR PAPER

# Integration of DFDs into a UML-based model-driven engineering approach

**João M. Fernandes · Johan Lilius · Dragos Truscan**

Received: 7 December 2004 / Accepted: 20 October 2005 / Published online: 20 June 2006
© Springer-Verlag 2006

**Abstract** The main aim of this article is to discuss how the functional and the object-oriented views can be inter-played to represent the various modeling perspectives of embedded systems. We discuss whether the object-oriented modeling paradigm, the predominant one to develop software at the present time, is also adequate for modeling embedded software and how it can be used with the functional paradigm. More specifically, we present how the main modeling tool of the traditional structured methods, data flow diagrams, can be integrated in an object-oriented development strategy based on the unified modeling language. The rationale behind the approach is that both views are important for modeling purposes in embedded systems environments, and thus a combined and integrated model is not only useful, but also fundamental for developing complex systems. The approach was integrated in a model-driven engineering process, where tool support for the models used was provided. In addition, model transformations have been specified and implemented to automate the process. We exemplify the approach with an IPv6 router case study.

**Keywords** MDA · UML · Data-flow Diagram · Model Transformation · Embedded Systems Specification · Process Model · Activity Diagram · IPv6 router

This article is an extension of two papers [1,2] published in the proceedings of IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2004).

J. M. Fernandes (✉)
Departamento de Informática,
Universidade do Minho,
Braga, Portugal.
e-mail: jmf@di.uminho.pt

J. Lilius
Computer Science Department,
Turku Centre for Computer Science
and Åbo Akademi University,
Turku, Finland.
e-mail: Johan.Lilius@abo.fi

D. Truscan
Embedded Systems Laboratory,
Turku Centre for Computer Science,
Turku, Finland.
e-mail: dragos.truscan@abo.fi

## 1 Introduction

In recent years, object-orientation constantly gained popularity over other specification techniques, becoming one of the main tools used for software development. Beside object-oriented methods, other languages have been proposed and used for similar purposes. One such language is provided by the structured analysis methods [3, 4] introduced in early 1970's, which became quite popular in industrial environments. Still, the structured analysis methods were largely overshadowed by the object-oriented design methods, especially after the introduction of unified modeling language (UML). Although, currently the convention is to use either a pure object-oriented approach or a "pure" structured approach, we prefer to view the two approaches as complementary, each one with its own strengths and weaknesses. Both object-oriented and structured analysis methods represent viable and necessary tools in embedded systems' design, each of them providing important techniques for describing the system under consideration. Moreover, as pointed out in [5,6], one

conceptual model is limited to represent only a specific view of a system, filtering out important details of the specification. Sometimes several views of the system under development are needed to capture all (or at least most) of its features and details. Ideally, the designer should be able to benefit from an approach where several design methods and conceptual models can be combined to specify the system at different abstraction levels [7].

Unfortunately, a culture of rivalry exists in the software community with respect to the two major paradigms. 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 [8–11], but none of them is widely used. Although some researchers [12] 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 very good results in some cases. Computing science seems also to benefit when opposite or dual aspects are taken into consideration. Indeed, significant improvement has always been achieved when the fruitful integration of a dual pair was possible [13]. That observation was a motivation for this work.

### 1.1 Comparing modeling methods

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 to abandon the old approach. 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 currently 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 [14]. But almost certainly we could make a similar claim in favor of structured methods if an adequate problem is used.

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" that shows that the object-oriented paradigm is definitely better than structured methods [15], and some authors even suggest the reverse [16]. In fact, attempts to prove formally that one approach is better than another are seldom effective, in any domain. This is especially difficult 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 the approaches.

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 [17]. In our opinion, there is not too much surprise in this fact, since object-orientation can 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*" [18].

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 [19]. 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.

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. The vast majority of object-oriented methods have the class diagram (a data-oriented model) as their main modeling tool, while structured methods use an activity-oriented model, i.e. data-flow diagram (DFD), as its principal diagram. DFDs use four symbols to represent any system at different levels of detail. The four modeling concepts to be represented are: data flows (movement of data in the system), data stores (repositories for data that is not moving), processes (transformation of incoming data flows into outgoing data flows), and external entities (sources or destinations outside the system boundary). DFDs provide an activity-based behavioral view of the system, useful for describing transformational systems, such as digital signal-processing systems, compilers, multimedia systems, or telecommunication devices.

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. However, for embedded systems an activity-oriented view of the system is typically more useful.

### 1.2 The functional and object-oriented views

The structured methods provide a functional view of a system, also designated as dynamic or behavioral, describing the perspective that centers around the behavior of the system. Similarly, the object-oriented view is seen as the perspective that focuses 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 [20].

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 [21].

A similar systemic view was proposed in [22]. 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 [23]. 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 specification language, as usually occurs. Hence, the following languages, 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 [24].

### 1.3 Related work

Many authors have already studied the combination of DFDs with object-oriented methods. Here we only discuss some approaches that are relevant for this work.

Within the object-process methodology (OPM), the combined usage of objects and processes is recommended [25]. An object-process diagram (OPD) can include both processes and objects, which are viewed as complementary entities that together describe the structure and behavior of the system. Objects are persistent entities and processes transform the objects by generating, consuming or affecting them. In addition, states are also integrated in OPDs to describe the objects. The usage of OPM, for modeling, specifying, and designing reactive and real-time systems, was also proposed, by extending the notation with notions such as timing constraints, events, conditions, exceptions, and control flow constructs [26].

In [27], the DFD notation is modified and the roles of the functional models are redefined, in order to use DFDs while retaining the spirit of object-orientation. Two types of functional models are suggested: object functional models (OFM) and service refinement functional models (SRFM). OFMs model the services provided by individual objects and SFRMs model how the services of individual objects can be composed to implement the services of their corresponding aggregation object. In both models, the only modeling elements are: objects, processes and data-flows. The data store is not necessary, according to the authors, since they use an object for that purpose. The interactions with a data store are modeled as communications with the corresponding object.

In another proposal [28], the functionality associated with each use case can be described by an E-DFD (an extended version of the traditional DFD) or an activity diagram, with the objective of automatically identifying the objects/classes of a distributed real-time system. E-DFDs are mapped into a graph and a tool automatically partitions the graph, which allows the identification of a set of objects that constitute the best architecture, from the design and test points of view.

In OMT, DFDs are used to describe the functional model of a system [29]. Since in OMT the system is also specified by two other models (the object and the dynamic models), DFDs specify the meaning of the operations in the object model and the actions in the dynamic model. But, although there is some attempt at integration, this correspondence is left completely vague and can not be analyzed in any useful way.

For reverse engineering purposes, the adoption of reverse generated DFDs (i.e., DFDs obtained after interpreting the source code) is proposed as the basis to obtain the objects that a system is composed of [30]. The approach is said to be hybrid, because it is not fully automatic, requiring in specific occasions the assistance of a human expert with knowledge of the domain. Again in a reverse engineering context, it is suggested the combined usage of DFDs and ERDs to describe the system being modernized [31].

Alabiso also proposes the transformation of DFDs into objects [8]. To accomplish the transformation he proposes the following activities: (1) interpret data, data processes, data stores and external entities in terms of object-oriented concepts; (2) interpret the DFD hierarchy in terms of object decomposition; (3) interpret the meaning of control processes for the object-oriented model; (4) use data decomposition to guide the definition of the object decomposition.

Another interesting proposal is the Functional and object-oriented methodology (FOOM) [9], which is specifically tailored for information systems. FOOM's main idea is to use the functional approach, at the analysis phase, to define users requirements and the object-oriented approach, at the design phase, to specify the structure and behavior of the system. In FOOM, the specification of user requirements is accomplished, in functional terms, by OO-DFDs (a DFD with data stores replaced by classes), and in data terms by an initial object-oriented schema, or an entity-relationship diagram (ERD) which is easily transformed into an initial object-oriented schema. In the design phase, the artifacts from the analysis are used and detailed object-oriented and behavior sche-mas are created. These schemas are the input to the implementation phase, where an object-oriented programming language is adopted to create a solution for the system.

From the approaches presented in this section we consider that [8,9,30,31] are the closest to our work, but they do not fit to what we consider is needed for designing embedded systems for the following reasons: (a) [30] and [31] are targeted to reverse engineering and the process of defining DFDs requires human intervention; (b) [9] addresses information systems, placing data and behavior on equal footing, while we want to put behavior first; (c) [8] is the closest to our approach, but it follows a structural approach in the object-oriented domain, while we would like to focus on the functionality of the system.

### 1.4 Contents and main contributions

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.

The proposed integration could look forced or unnatural, because we are trying to unite two apparently discordant approaches for developing software systems. Nonetheless, software engineers should not take a religious or dogmatic attitude when it comes to a specific model. Currently the question that must be answered by the software engineers is how to nicely integrate different models, if all of them are deemed valuable for the description of the system [7]. This is an important question today when commonly adopted graphical modeling languages such as UML include several diagrams that are only loosely related. The proper integration of theories and concepts is one of the key challenges in the field of embedded systems [32].

The main motivation for this comes from a case study [33], where we have applied object-oriented techniques and UML to the design of a protocol processor. In this case study, we noticed that there was a functional and structural view of the system that was not adequately represented by the diagrams provided in UML. Therefore, the integration of both DFD and UML appears to be a solution for this question. Additionally, industry currently needs to cope with high levels of design complexity, that can be tackled using model-driven approaches to provide automation and consistency checking during the analysis and design phases of embedded systems.

The paper also proposes a practical approach in combining the DFD and UML notations as part of a model-driven engineering process. Due to the increasing complexity of current embedded systems, model-driven development has become one of the prerequisites for systems development. The main idea behind a model-driven approach is the usage of well-defined models to represent abstract specifications of the systems. The models are built following rules specified by their modeling language (i.e. meta-model). To exploit the possibility of automation provided by models, appropriate tool support is required, allowing the designer to navigate among different views of the system in a tool supported and automated manner. In this paper, we focus on the analysis phase of the development process.

The paper is structured as follows. In the next section we present our opinion on the usage of UML in the context of embedded software. Then, in Sect. 3 we provide an argumentation on the necessity to use both UML and DFD for the specification of embedded systems and we discuss possible solutions to accomplish this integration. We present in Sect. 4 our model-driven engineering (MDE) process based on UML-DFD integration. We show how we modeled the different views of the system and how we specified model transformations between the views. Section 5 presents the tool support for the MDE process and we give examples of how the model transformations were automated using scripts. Finally, Sect. 6 presents concluding remarks and points out some future work. It is important to notice that the UML version under consideration in this paper is 1.4.

## 2 UML for embedded software

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 popularity and usage is the UML. When UML was created it missed process, data and user interface models [34]. As it stands today, UML 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 [7].

### 2.1 Typical usage of UML

The most common way to use UML diagrams during analysis is to start with use case diagrams for capturing the user requirements. For modelling business processes [35] use cases with some stereotypes can also be used. This solution can be complemented by using an activity diagram that shows how use cases are related, as well as alternatives and decisions for them [36, p. 51]. Next, sequence diagrams are used 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 five 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, while component and deployment diagrams are not at all used or only used in later development stages.

On of the problems with this typical usage, in what concerns the development of an embedded system, is that 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 [32].

### 2.2 Classes versus objects

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 [37]. We provide in this section arguments on the importance of the object diagram for embedded systems development.

The typical focus towards classes may be caused by the fact that many software engineers still do not clearly distinguish between objects and classes [38,39]. This confusion is greatly related to the intangible nature of software. In the same time, the way the UML meta-model is defined imposes a certain sequence in which the diagrams are created. For instance, one cannot create an object diagram unless the class diagram containing the class specification of the objects was created in advance.

For conventional software, the class diagram is built first, but we believe that for embedded software, that order must be reversed. To develop embedded software, it is more important to have a good object model than a good class diagram, because the elements that do constitute the system are the objects and not their classes [40]. Obviously, the best situation is having good object diagrams and good class diagrams. (But, as a guideline for development, we consider that classes and objects should not be simultaneously incorporated in the same model.) Therefore we recommend, when develop-

ing embedded systems, to first identify the objects and to later select the classes which those objects belong to.

The emphasis on objects (instances) is justified since we are dealing with real-time embedded systems. This object-driven or component-based approach is typically followed for developing control-oriented systems, where the final architecture and the concepts of abstraction and modularity are key topics to guarantee that the non-functional requirements (heterogeneity, ubiquity, fault-tolerance, security, dependability) are met. In contrast, class-driven approaches are usually used for information-intensive applications, such as databases, in which the relations among classes (types) and its hierarchical categorization are the most important issues to consider.

This perspective that puts classes in an apparently secondary role may be classified by some specialists as object-based rather than object-oriented. However, the approach that first defines the objects and later the classes is somehow consistent with the bottom-up discovery of inheritance to organize the classes [29, p. 163].

The class diagram is usually understood as a template for a set of applications that can be obtained from it. In other words, the class diagram is a high-level generalization of the system [41]. When developers define the way classes are interrelated, they are indicating all the systems (or all the configurations) that can be obtained from those classes.

With this perspective, it is common not to build the object diagram, since it can be automatically derived from the class diagram. In the cases where an object diagram is built, it is mandatory to guarantee that the relations expressed in the class diagram between two classes also exist between instances of those classes. This is the main reason why methodologies usually suggest class diagrams to be collaborate first constructed, rather than object diagrams.

The class-centered approach seems adequate to develop business information systems or, more generally, any data-dominated system, where the objects are created and eliminated during the system life cycle [40,42]. For example, in a system for bank accounts management, it is common that each account is always associated with, at least, one customer. This fact is indicated in the class diagram by associating the account class with the customer class. When an account object is created, it must be linked to, at least, one customer object. This approach does not offer many benefits for developing embedded systems, since at the highest levels of abstraction the objects that constitute the system are not created and destroyed on the fly.

Actually, structure is one of the dominant aspects of real-time and embedded systems [42]. An embedded system is generally composed of a set of fixed objects that are linked in some way and this organization can be perfectly described by an object or a collaboration diagram. Thus, it is not crucial to indicate, for example, that objects of the controller class need to be linked with objects of the sensor class, because this fact is not at all universal. If in some applications this information can be important, it may be completely wrong in others.

Another important concern related to embedded software is the description of concurrency. The notion of concurrency can also be modeled in UML using objects, namely active ones [42]. An *active object* is continuously executing, which requires its own thread of control, and runs concurrently with other active objects. The 'Embedded UML' profile proposes also the concept of a *reactive object*, which consists of a concurrent process, with asynchronous communications to other objects, that reacts to external events and stimuli [43]. A reactive object is one that can react to events and, therefore, must specify a control structure, typically in the form of a state-oriented model, and communication with other objects through ports and connectors, which gives raise to collaboration diagrams. So, objects do play a fundamental role in modeling embedded software systems.

## 3 Combining DFD and 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, where several possible alternatives are considered. 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. Yet 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 useful models for embedded software?
2. In which phase of the development process must DFDs be introduced?
3. Which views should DFDs cover?

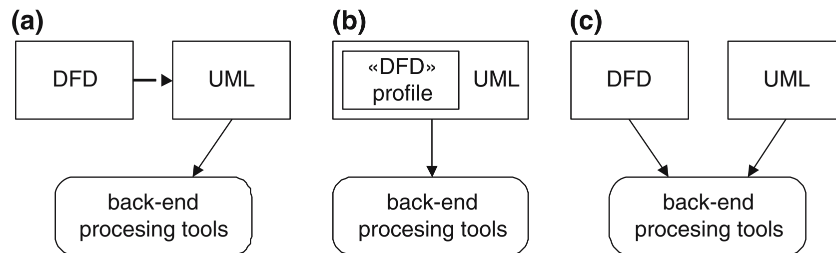



**Fig. 1** Three alternatives for integrating DFDs in UML: **a** DFDs mapped into UML concepts; **b** DFDs added to the UML meta-model, i.e. profile; **c** DFDs as a separate meta-model coexisting with the UML meta-model

### 3.1 Initial considerations

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. 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.

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 [44] and control/data flow graphs (CDFG) [45]. 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 [4], 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 solution, but that in others they are not.

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 three approaches (Fig. 1).

1. The DFD meta-model is mapped into UML concepts, where DFD elements are represented using already existing UML based on some convention of interpretation;
2. DFDs are represented as an extension of UML (i.e. profile) that coexists with the UML models;
3. Both UML and DFD meta-models are available as originally devised.

Each of the alternatives brings its own advantages and disadvantages. Although the first alternative brings the benefit from using existing UML tools without additional work, it does not provide mechanisms to enforce the well-formedness of models with respect to the DFD meta-model. Thus it is entirely left to the designers to decide what UML elements best represent DFDs. The second alternative defines a UML profile for DFD, where by extending the UML core, one can still benefit from the UML tools and notation, but some constraints on the well-formedness of models can also be provided. In the third approach, the DFD and UML meta-models are specified independently but they may coexist inside the same tool. Still, one can benefit from the specific DFD graphical notations, but their implementation requires additional work. Having a well-defined meta-model for DFD supports better the well-formedness of the models, much in the same way as in UML. In fact, using the OMG's meta-object facility (MOF) [46] one can reuse parts of the UML meta-model core to create a meta-model for DFD. To follow this approach, a MOF-based meta-model for structured analysis for real-time (SA/RT), was defined and built in [47]. The meta-model was specified so that it can function perfectly well as a stand-alone tool, as well as in combination with other MOF-based meta-models, providing the possibility to integrate DFD models with UML tools.

The alternatives presented can be also applied when integrating other meta-models with UML. We believe that although the first two alternatives are preferable in certain situations (e.g., defining a UML profile for an object-oriented language like SystemC), by allowing us to restrict to the UML meta-model in what concerns the model's back-end processing (model transformation, validation, code generation), in other situations (e.g., domain-specific languages, non-object-oriented hardware description languages, etc.) the third one is the only solution. Restricting the UML meta-model 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 set of 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. 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.

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

1. The DFDs to refine the use case model;
2. The DFDs to detail the behavior of a system's component;
3. The 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.

Here, we will concentrate on the first proposal, and only shortly discuss the others.

### 3.2 DFDs to refine the use case model

Use cases are among the modeling techniques the developers can rely upon to analyze their systems. Thus, it is commonly accepted, within the object-oriented community, that the analysis of a software system can 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. Yet, use cases are not intended to be used alone, but to be complemented with other techniques (e.g., user stories, sequence diagrams, scenarios, etc.).

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 an object-oriented approach [48]. 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 always incorporate object-oriented diagrams in the modeling process.

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), 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. It is usually difficult to perceive how use cases interact, especially if 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.

We would like 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 that of the DFD. However, this choice should be taken with care, since different diagrammatic representations do not necessarily have the same effectiveness or computational power.

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 matter 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 [49]. With this view, 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 (called *functional primitives*) must be associated with a description, traditionally a PSPEC (process specification), that specifies concisely the intended behavior. In fact, when a system is divided in parts both structure and behavior are being decomposed. For example, Kiczales et al. [50] 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. They claim 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

is acceptable that we consider that all those design methods force equally the systems to a structural decomposition.

Our opinion is that *collaboration diagrams* (or *communication diagrams* in UML 2.0) 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.

### *3.2.1 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, and faces several problems. Firstly, despite the existence of some proposals for automatically obtaining objects, namely the SysObj tool [28], 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.

To tackle these crucial questions, namely the identification of objects from use cases, some proposals exist [52–54], 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 [6,40].

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) [52]. 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 (e.g., an object to provide a GUI, an object to interface a sensor, etc.) models behavior and information that depend on the system's interface, that is, the dialogue of the system with the actors that interact with it. A data-object (e.g., a bank account) predominantly models information, whose existence must be lengthy. 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. 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 most probably 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 contradict, for control-objects to usurp the responsibilities of data-objects [55]. 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.

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 certainly be used, 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*" [51].

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.

[1] These three categories can also be designated as boundary, entity, and function, respectively.

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. DFDs should not be used to model the details of the information perspective of the system, since other diagrams are used for that purpose.

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).

### 3.2.2 Possible enhancements

The 4SRS can be enhanced in several directions, and here we intend to provide one of these possible improvements. The idea is that the transformation of each use case into objects can be eased if the use case is classified, according to some scheme. This classification would provide some hints on which object categories to use and how to connect those objects.

To understand what the classification mechanism can be, we must first study how many combinations of objects of a given use case do exist. The 4SRS assumes that each use case gives rise to a maximum of three objects:[3] one interface-object (i), one control-object (c) and one data-object (d). Thus, we come up with 8 different combinations (Ø, i, c, d, ic, di, cd, icd), if we ignore the links among objects. These combinations can be arranged hierarchically, as illustrated in Fig. 2.

If we now take in consideration the links among objects, they are trivially established for the combinations of c0, c1 and c2 levels. Actually, only the c2 level combinations have one link, since for the others there is only one object or no object at all. For the c3 level combination, we have several possibilities for the links. In all of them a minimum of two links must exist, since it is

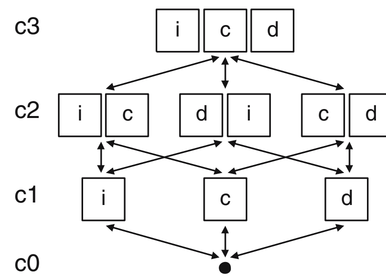



**Fig. 2** Combinations of objects

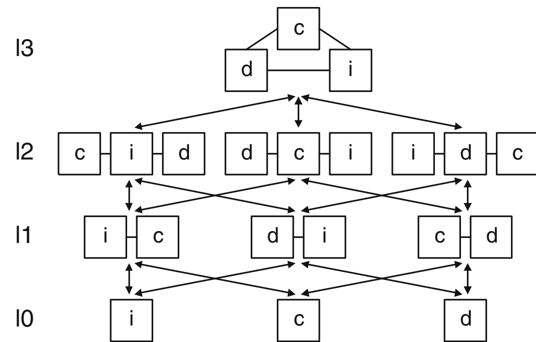



**Fig. 3** Combinations of linked objects

assumed that the objects must be fully-interconnected, at least indirectly. All these combinations with links, depicted in Fig. 3, can be represented as: i, c, d, i-c, d-i, c-d, c-i-d, d-c-i, i-c-d, -i-c-d-. This last symbol is intended to represent the fully-interconnected object combination at level l3. It must be stressed that the links between two objects do not necessarily represent software messages, but rather indicate only a logical association.

Some of the combinations in Fig. 3 might not make sense, from a modeling perspective, namely if the semantics associated with DFDs are taking into account, which excludes, for example, two data stores directly connected. Some restrictions may also apply to the object diagram, if the rules for robustness diagrams presented in [53, p. 69] are followed (Fig. 4). These rules presuppose a more restrictive view on the categories of objects than our perspective, but they may apply nicely and efficiently in some contexts. The first observation is to include always, for each use case, an interface-object to communicate with the actors, which means that an interface-object is supposed always to exist. This is not the case only for the situations where internal use cases (i.e., internally-initiated functionalities) exist. The rules also disallow an interface-object to be connected to a data-object.

The robustness diagram rules can also help the developers to link objects originated from different use cases. Apparently, no restrictions apply to control-objects, since they can be connected to all the categories of objects.

[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.

[3] In some situations, four or more objects can be created from the same use case. A typical example is the creation of two interface-objects, one for input and the other for output purposes. However, considering three as the maximum number of objects does not result in loss of generality.

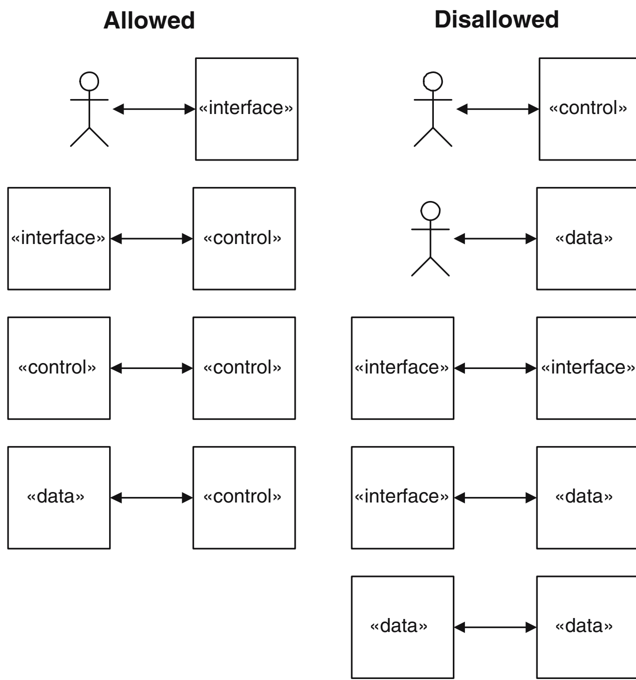



**Fig. 4** Robustness diagram rules (adapted from [53])

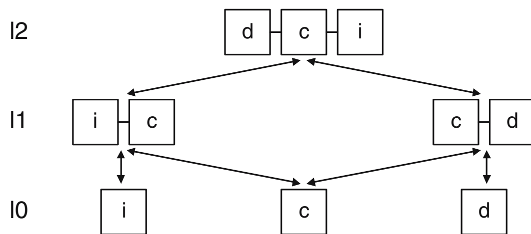



**Fig. 5** Combinations of linked objects, taking into account the rules for robustness diagrams

In principle, two data-objects should not be directly connected, since they are seen as passive objects. Interface-objects and data-objects should only be linked to control-objects. Figure 5 is a rearrangement of Fig. 3, but taking into consideration the rules for robustness diagrams, which result in deleting some of the combinations.

The main idea behind this approach is thus to classify the use cases according to some criteria and with that classification have more information on what object configuration (i.e., software architecture) is more likely to give an appropriate computational support for the use cases.

## 3.3 DFDs to detail the behavior of a component

In this paper, we explore very shortly this hypothesis of using DFDs. This possibility was already suggested, for example, by Ivar Jacobson in a conference panel [12]. Briefly, we can comment that the UML meta-model defines an association between *ModelElement* and *Statemachine*, called *behavior* [56, 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 [56, 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 or a Signal.

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, we must 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, by distributing the different system's views across several diagrams.

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

## 3.4 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 transform the program code into DFDs and other complementary models, than to go directly into some object-oriented models.

We propose that DFDs could be transformed into object or class diagrams. We give more details, namely algorithms, about these transformations in Sect. 4.5. Some similar ideas were already proposed in the FOOM methodology [9] for developing information systems, but its usage for embedded systems requires 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 [11].

## 4 A model-driven engineering approach

In this section we present our model-driven engineering approach, where we integrate the object-oriented and data-flow paradigms from a modeling perspective.

Recently, OMG started to promote a new vision to system development: model driven architecture [57], where the main emphasis is put on separating the development process from the implementation one, making use of models to describe the system at different steps of the development. The main modeling tool of MDA is, unsurprisingly, represented by UML. But the way MDA is envisioned gives the possibility of combining several modeling languages, and consequently their paradigms. As previously presented, sometimes several views, some of them not supported in UML (thus belonging to a different meta-model) have to be used to obtain a good understanding of the system.

In the previous sections, the motivation for combining the object-oriented and functional views of the systems was given and a number of possible ways to integrate the two were proposed. In this section we will present our practical approach in combining the two paradigms from the perspective of a model-driven philosophy. In this sense, a set of models of the system, a process for integration of these models and a number of model transformations to go from one model to another are defined. The method we present focuses on extracting the behavioral specification of the system (i.e., an IPv6 Router) corresponding to the realization *platform independent model* (PIM) in the MDA specification. The process "refines" several PIM models until the necessary level of detail is reached, before proceeding to the PIM-to-PSM (*platform specific model*) transformation. The main purpose of the IPv6 Router case study is to serve as a feasibility demonstration of our (MDE) approach.

The basic idea behind our approach is to specify the system following a functional decomposition and representing it using the benefits of both object-oriented and DFD views. Some of the ideas presented here were already analyzed and discussed in [33], but with a different perspective. There, the main objective was to define a complete UML-based methodology for embedded systems, making special emphasis on the real implementation of the system. Here, our aim is to provide a practical way of merging DFDs with other models (e.g., object or class diagrams), during the analysis phase and to develop a viable tool support for creating and maintaining these models during the entire life-cycle of the design. The main phases of the process (Fig. 6) are:

(a) Extract application requirements
(b) Create the use case diagram (UCD)
(c) Specify each use case in a textual manner
(d) Transform the UCD into an initial object diagram (IOD)
(e) Refactor IOD by grouping, splitting and discarding objects based on their functionality
(f) Transform the IOD into a DFD
(g) Identify data flows and build a data dictionary
(h) Specify process behavior using activity diagrams
(i) Transform the DFD into an object diagram (OD) `or`
(j) Transform the DFD view into a class diagram (CD)

During the design flow we have to change several times the view of the system to be able to work on specific details provided by each view. To provide an automatable approach, well-defined model transformations are required. Several authors recognize the model transformation as the fundamental mechanism for model-driven development [58]. A model transformation takes a source model expressed in a given language and transforms it into a target model expressed either in the same language or in a different one. We consider that model transformation is an important technique for applying software patterns and refactorings, in the same time providing a good support for reuse. Following, we briefly describe the algorithm for the main steps in Fig. 6, exemplified with an IPv6 router specification.

### 4.1 Capturing the requirements with use cases

We start by analyzing the specification of the IPv6 router requirements and building a use case diagram (Fig. 7). Two external actors interact with the router. The `node` is a common network node that requests the router to forward datagrams and eventually to send back ICMPv6 error messages in case of failure. The `router` represents the neighboring routers that exchange topological information with our router. Then, we identify the services that the system provides to the external environment and extract them into a list of use cases. We have identified six use cases that provide services for external actors. Each use case is accompanied by a short textual description that specifies its functionality. Due to space reasons, we intentionally omit technical details of the router specification. More details can be found in [59].

As discussed in Sect. 3.2 use cases represent a technique that allows us to follow either a structured or object-oriented approach. Indeed, next we show how we obtain a DFD starting from a use cases diagram, by first identifying the initial objects and then creating the IOD of the system.

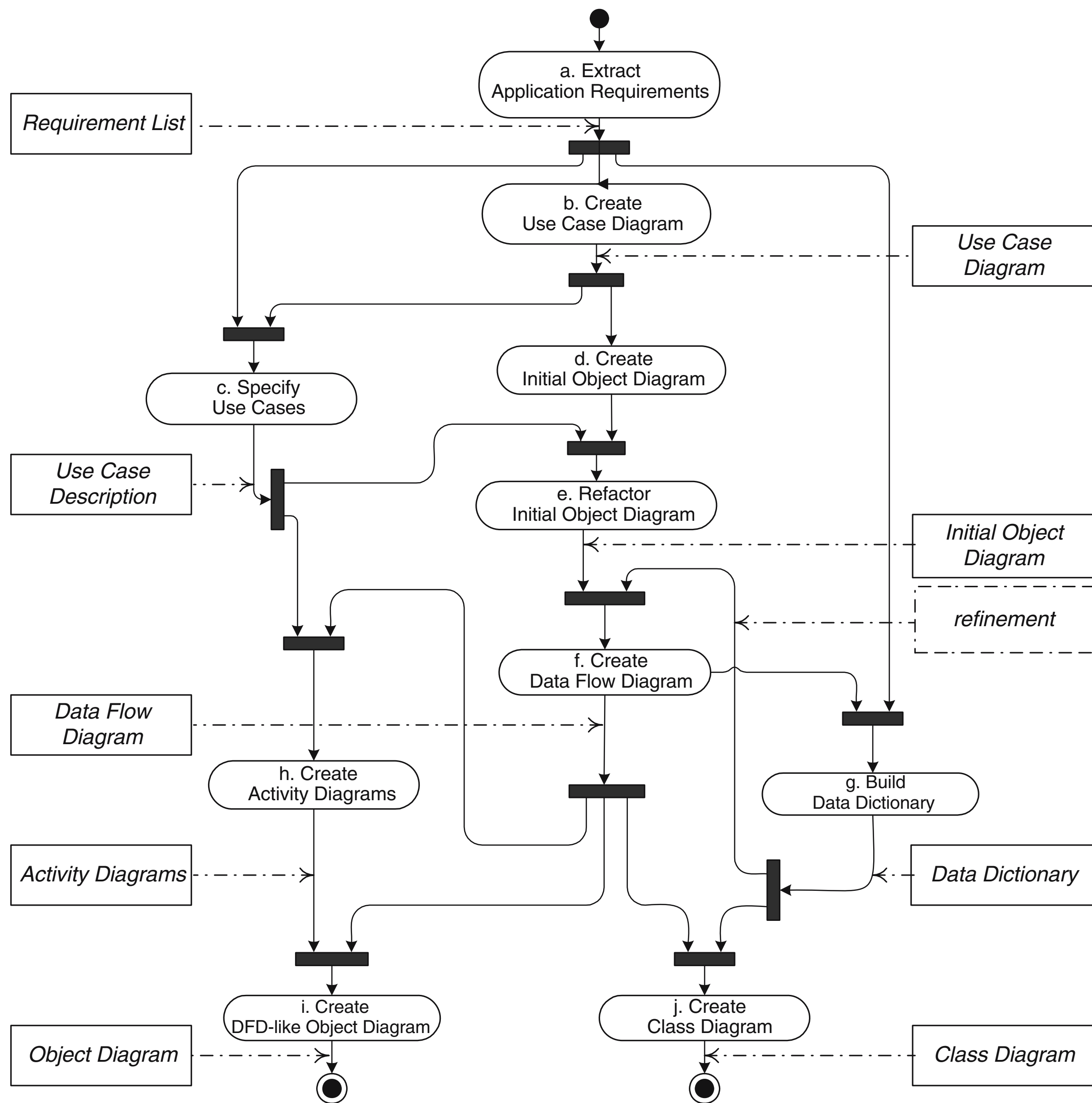



**Fig. 6** Activity diagram describing the process for integrating UML and DFD models

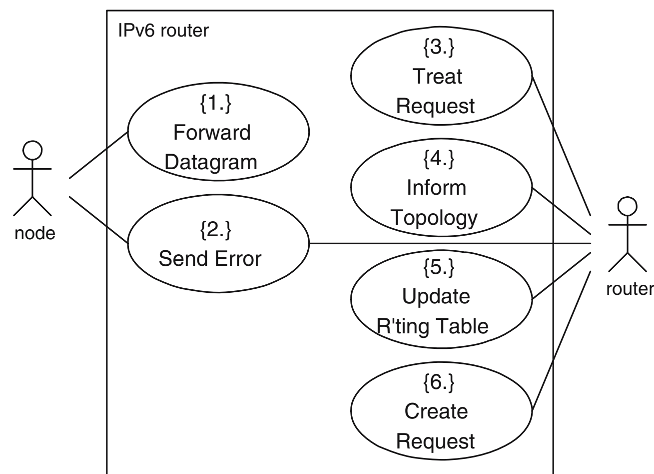
## 4.2 From use cases to initial object diagram

From the use case diagram (UCD) we identify the initial set of objects in the system by decomposing each use case into three objects: control, data and interface-objects, based on the 4SRS method. The objects have the same number as the initial use case, but an extra tag name is added. For instance, the {1.i}, {1.c} and {1.d} objects (Fig. 8) represent the interface, control and data-objects obtained from splitting the `Forward Datagram {1.}` use case. Also, an actor is created in the initial object diagram (IOD) corresponding to the actors in the UCD.

The model is structured so that all communication between data-objects and interface-objects is done through con-trol-objects. Consequently, we add associations between interface and control-objects, and between control- and data-objects, respectively. We also add associations between objects and the external environment (actors).

When a system is divided into parts, both structure and behavior are being decomposed along with functionality. Usually the designer is focusing his effort only on one view during decomposition, but we consider that the other aspects are always present as side-effects.

**Fig. 7** Use case diagram for the IPv6 router



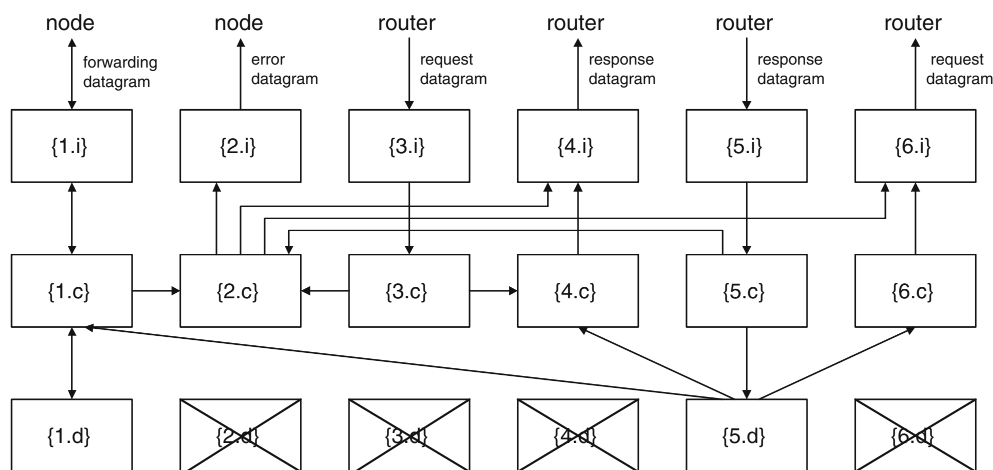




**Fig. 8** Initial object diagram for the IPv6 router

The main steps of transforming the UCD into IOD are:

1. Each actor in UCD is transformed into an actor (instance) in IOD
2. For each use case in UCD, three objects (interface, control, data) are created in the IOD
3. In IOD, an association is drawn between the interface and control-objects belonging to the same use case
4. Similarly an association is drawn between control and data-objects belonging to the same use case
5. For each actor–usecase association in UCD, an association is drawn between the corresponding actor and the interface-object generated by the use case.

### 4.3 Refactoring the initial object diagram

According to the 4SRS method, by instantiating scenarios for the initial use cases, we decide what objects are kept or disposed, and also identify the communication (depicted by associations) among these objects. The remaining set of objects is refactored, by decomposing or grouping together objects of the same category. The interface-objects that communicate the same information with the same actor and in the same direction can be grouped together. For instance, objects {5.i} and {3.i} receive both routing datagrams, either as `request` or `response datagram`, from the router. An interface-object that communicates bidirectionally with one node, can be split into 2 separate interface-objects ({1a.i} and {1b.i}), one dealing with the incoming and the other with the outgoing traffic. The final result of the refactoring is presented in Fig. 8.

The only data-objects that survived are {1.d} (i.e., `data-gram`) and {5.d} (i.e., `routing table`). Although we could also have the data-objects for use cases {2}, {3}, {4}, {6}, we decided that only the major functionalities of the system ({1.} forwarding and {5.} routing) should have a data object. This does not mean that information is not produced in objects {2.c}, {3.c}, {4.c} and {6.c}, but that the major role of those objects is to produce behavior.

The links between {5.d} and {1.c}, {4.c} and {6.c} represent the dependency of those control-objects on the routing information, which is stored in object {5.d} (i.e., `routingTable`).

Although, splitting and grouping of objects is tool supported, the refactoring process is, for the moment, performed manually based entirely on the designer's expertise.

## 4.4 From object diagram to DFD

Sometimes designers need to change the view of the system to be able to focus on different details. We use DFDs to identify, classify and refine data flows involved in the system. We obtain the data flow model of the system from the initial set of objects (Fig. 8), by noting the direction of the communication among the objects and then defining data entities that are being exchanged. To obtain the DFD of the system two steps are required: to specify the processes (*data transformations*) and *data stores* in the system, and then to identify the *data flows* connecting them.

The first step of the process is straight-forward due to the way the IOD was structured. In the IOD we have control and interface-objects that process input communication from external environment (i.e. `router` and `node`) and transform it into output communication. This is similar to the behavior of the processes provided by the DFD concepts, allowing us to transform all control and interface-objects into DFD-specific processes (i.e., *data transformations*). Similarly, data-objects in the IOD are transformed into *data store* elements in DFDs.

Most of the DFD approaches in literature, do not make a clear distinction among processes with respect to their execution in time. Our opinion is that we can observe two types of behavior: processes that start their execution when one of their input flows becomes active, and processes that execute continuously, independent of their input flows status. We name them *reactive processes* and *active processes*, respectively. A process is considered to be active if it has no input flows from other processes or its behavior is self-triggered (output flows are fired without an input flow triggering the process). One example of an active process would be a process that is periodically reading a data store (e.g., processes {1b.c} and {6.c} in Fig. 9). Here, the input flows are triggered by the processes themselves and not by the data stores, despite the fact that there is a data-flow from the data stores to the given processes. Classifying processes into active and reactive helps the designer to specify, in the next design phases, the internal behavior of each process.

To transform an IOD into a DFD we perform the following steps:

1. Transform each actor in the IOD into an external entity
2. Transform interface and control-objects into processes in the DFD
3. Transform data-objects into data stores
4. Transform associations between elements of the IOD into data flows either between external entities and processes, or between processes and stores, or in-between processes
5. Identify and mark active processes in the DFD.

In the second step, we focus on refining the previously added data flows, by adding new details about the data entities transported by *data flows*. The resulting diagram is presented in Fig. 9. We classify the data flows involved in the system by building a *data dictionary*. This is done by analyzing the data that is moved between data transformations, having as primary information source the application requirements. Data flow identification is performed manually and it is based, for the moment, on designer's skills. A complete data dictionary specification of the IPv6 Router under study can be found in [1]. Following we only present a small example, where the datagram types transported through the system (router) are classified.

```
Datagram = ForwardDatagram|RoutingDatagram|ErrorDatagram
    ForwardDatagram = IPv6Header+Payload
    RoutingDatagram = IPv6Header+UDPHeader+RIPMessage
    ErrorDatagram = IPv6Header+ICMPv6Message
```

## 4.5 From DFD to UML

For obtaining an object-oriented model of the system starting from the DFD-model we have tried two approaches. In the first (Sect. 4.5.1), we transform the DFD model directly into an object diagram by mapping (on an one-to-one basis) processes and data stores in the DFD model into objects. The second approach (Sect. 4.5.2) follows a more object-oriented view, where we focus on classifying data in the system and detecting class methods that operate over identified data.

### *4.5.1 From DFD to object diagram*

To obtain the object diagram OD, basically, the algorithm transforms each *data transformation* in the DFD into an *object* in OD, and the *data flows* among these transformations into *associations*. In addition, the data flows involved in the system become internal attributes

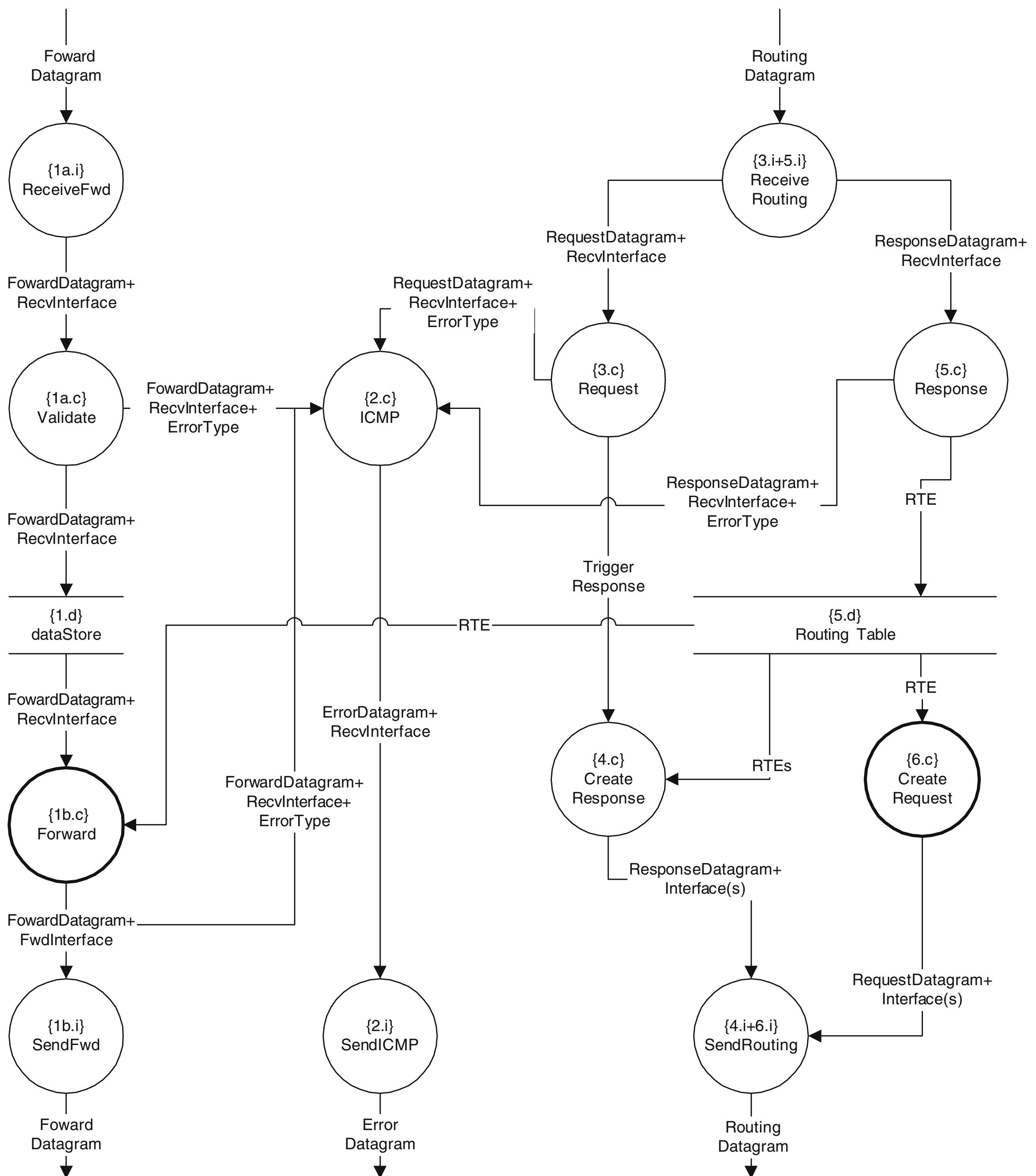



**Fig. 9** Data-flow diagram for the IPv6 router. Active processes are drawn in *thicker lines*

of the classes (objects) are encapsulated into the objects and used now to describe their internal state. In order to access the data entities inside objects, corresponding methods are added. For instance, the initial `ForwardDatagram` data flow, between the `ReceiveFwd` and the `Validate` processes (Fig. 9), has been encapsulated inside

the `Validate` object (Fig. 10), and it is only accessed by the `sendFwdDatagram()` method, while its value is dispatched to the adjacent objects through the `write-FwdDatagram()` and `sendFwdError()` methods.

In addition, objects originating from processes placed at the border of the system will have `set()` methods to communicate with the environment, while the other processes will have `send()` methods that receive input data flows in the DFD as parameters. Objects obtained from active processes will have in addition a `run()` method specifying their state machine, while the objects originating from data stores receive `write()` and/or `read()` methods to provide access to their data, based on data flows accessing them. One should note that the way the elements in the object diagram are named is only to help in automating the process. Once the transformation process is completed, the names of different elements can be changed to be more meaningful for the designer.

When the transformation is completed, we focus on specifying the behavior of each object. In fact, defining the behavioral aspects of the system is the main purpose of this view. The final object model (Fig. 10) is very similar to the DFD one, but now we have objects that have an internal behavior and provide services (implemented by methods) to the adjacent objects. The newly created objects are also classified as being active or reactive based on the processes from which they were generated. The difference between them is that the execution of the reactive objects is triggered only when one of their methods is invoked, while the active objects have a state machine (usually implemented by a `run()` method) executing continuously.

To summarize, the following steps are performed:

1. External entities in DFD are transformed into actors in OD
2. Processes in DFD are transformed into objects in OD
3. Flows between processes become associations
4. Active objects receive a `run()` method
5. Classes originating from border processes dealing with input communication receive an attribute corresponding to the input flow and a `set()` method to access that attribute
6. Objects originating from non-border processes receive a `send()` method of the incoming dataflow and the corresponding attribute
7. Data stores are transformed into objects, that contain `read()` and `write()` methods to provide access to their attributes.

[4] We mention that, due to typographical reasons, some details (i.e. attributes, actors, etc.) have been intentionally omitted from the diagram.

The object diagram in Fig. 10 provides a low level of abstraction and data encapsulation, but it proved to be suited for prototyping purposes and functional testing of the specification. Additionally, it is a good candidate for being mapped onto a hardware-based platform, because its granularity is at a relatively low level of detail.

We used this approach to design protocol processing applications targeted to the TACO processor platform [33]. The TACO processor is composed of functional units that communicate via an interconnection network of data buses. Resources of the processor are specified and implemented in a library of components using the SystemC language [60] (an object oriented extension of C++ for hardware specification language, where the hardware modules are instances of SystemC classes). Due to the architectural aspects of the processor a object diagram like the one presented in Fig. 10 proved well-suited for the specification and design purposes. To configure TACO to support a given application one has to select a number of required resources from its SystemC component library. Thus, being able to go from a structural representation to an object-oriented one at any point during the design flow proved to be helpful.

### *4.5.2 From DFD to class diagram*

In the second approach to create an object-oriented model of the system, we adopt a view where data involved in the system plays a central role. This approach is not far from the structured methods philosophy where determining the type of data involved in the system is the main task. The transformation between the models is based on the classification and encapsulation of data into classes, along with the corresponding methods that operate over this data.

Briefly, the algorithm classifies all the data flows and data stores inside the DFD, based on their type. For each identified type in the DFD, a corresponding class is created in the class diagram. In order to add class methods we look for three kinds of patterns in the data flow diagram: data flows communicating with the external environment of the system (Fig. 11), data flows between two processes (Fig. 12) and data flows that communicate with data stores (Fig. 13).

A number of processes (i.e., data transformations) operate over each data flow class inside the DFD. We transform these processes into logical methods of the classes corresponding to data flows they process. For instance, `Forward-Datagram` data flow (Fig. 9) is processed by different data transformations (`ReceiveFwd`, `Validate`, `Forward`, `Send-Forward`, `ICMP`). Thus, we create the `forwardDatagram` class inside the class diagram (Fig. 14) and we add the DFD processes that affects it

**Fig. 10** The object diagram of the IPv6 router[4]

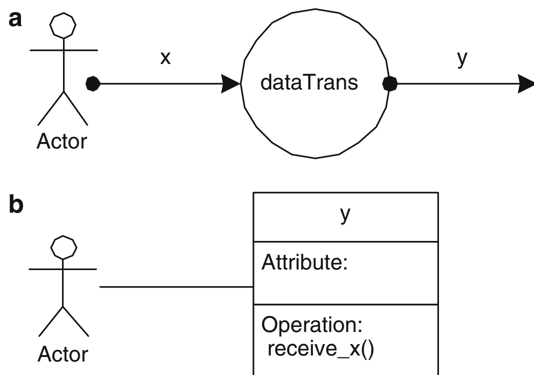



**Fig. 11** Border process pattern

as logical methods of this class. We consider that a process becomes a method of a class only if it has as output flow the data flow type represented by that class. For instance, the `ICMP` process in Fig. 9 is not transformed into a method of `forwardDatagram` because it outputs an `ErrorDatagram`.

Data stores in the DFD receive a special treatment. Since they are places that store data and our goal is classifying data in the system, each data store element is transformed into a separate class. The newly created class provides `read()` and `write()` methods to access data, based on the input and output flows of the initial data store (see `data Store` in Fig. 13). In addition, we classify data inside "Data Store" classes based on the data flows that access the Data Store element.

Here we also have a possible classification of classes into active and reactive. Active classes have the internal behavior described as a continuous running state machine that, based on the events it receives, can invoke methods of its own class or methods of the neighboring

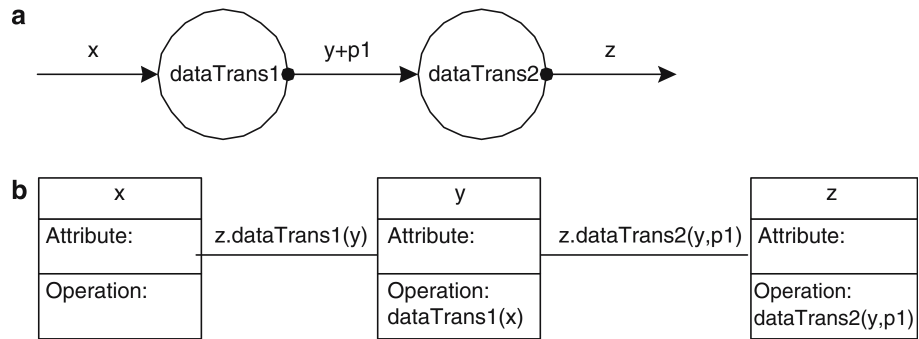



**Fig. 12** Interprocess communication pattern

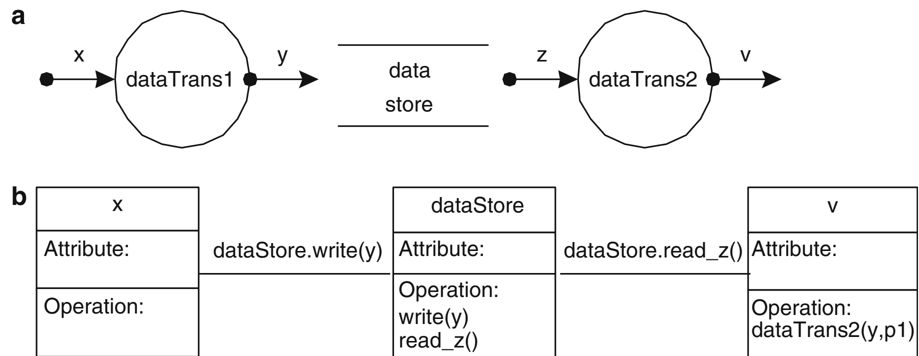



**Fig. 13** Data Store communication pattern

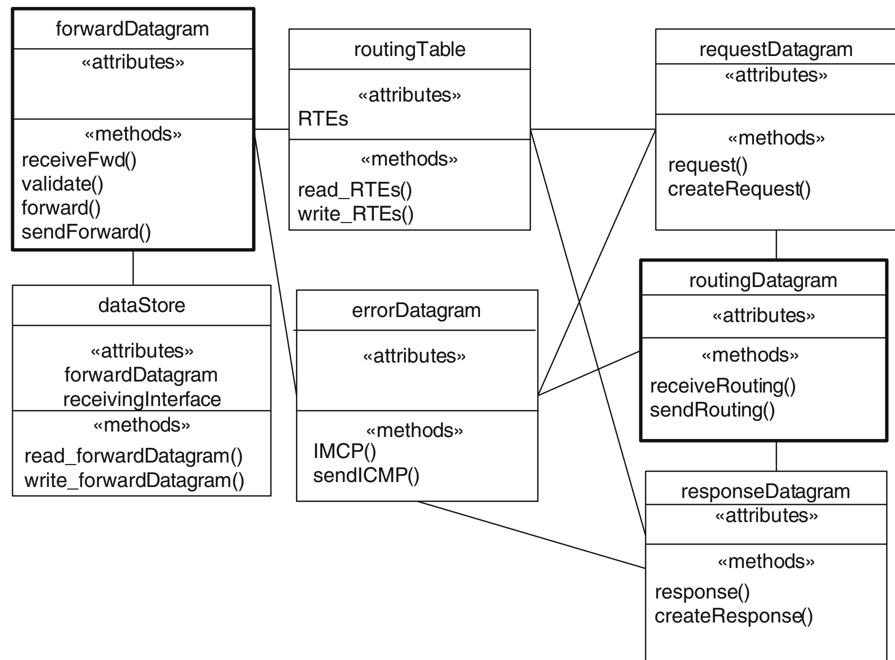



**Fig. 14** The class diagram of the IPv6 Router

classes. For instance, two active classes are present in Fig. 14, `forwardDatagram` and `routingData-gram` that were respectively derived from the processes {1b.c} and {6.c} in Fig. 9. The behavior of this classes is described by a state-oriented model (statechart or activity diagram). The methods of the class are being called internally (by the instances) when a specific state is reached. The methods are not called by other classes from the outside. For instance, the `receiveFwd` method is called when an event (e.g., `new_message_available`) is triggered in the

state-machine. Classifying classes into active and reactive is for now an ad-hoc approach, but we intend to further investigate it.

In brief, the transformation performs the following steps:

1. Transform distinct data flows and data stores into classes
2. Apply the Inter-process communication pattern
3. Apply the border process pattern.
4. Apply the data store communication pattern.

As a case study, we used the approach to generate Java code starting from the class diagram in Fig. 14. Due to the object oriented principles implied by the class diagram the code generation process was straight forward. The obtained code provided a simulation model of our specification that could be used as executable specification to check the functionality of the router. The object diagram in Fig. 10 could have been used for the same purposes but with some necessary adaptation.

As a result of implementing the router specification both in Java and on a protocol processor, we could conclude that both approaches can be equally followed. The first approach seems to be more natural for component-based hardware systems, while the second one fits well in object-oriented software-intensive designs. We consider that the designer should follow the one that fits best with his affinities, experience and working culture.

## 5 Tool support

To take full advantage of a model-driven approach, appropriate tool support is required. On one hand, tools should provide means to (graphically) create, edit and manipulate model elements. On the other hand, scripting facilities to implement automated manipulation and consistency verification of such models have to be provided, in order to speed up the design process and to cut down development times.

### 5.1 SMW toolkit

To create and manipulate the UML and DFD models, we make use of the Software Modelling Workbench (SMW) tool available for download at URL: `http://mde.abo.fi/confluence/display/SMW`. The tool is built upon the OMG's MOF and UML standards, allowing editing, storage and manipulation of meta-models. SMW uses the Python language [61] (an interpreted object-oriented programming language, with a slightly different syntax than C++ or Java) to describe the elements of a meta-model, each element being represented as a Python class. This fact provides the basic scripting mechanisms to query and manipulate models. Moreover, making use of the lambda-functions that the Python language provides, OCL (Object Constraint Language)-like idioms are supported. OCL is a semi-formal language, developed by OMG, to add more precision to the UML meta-models, beyond the capabilities of the graphical diagrams.

SMW allows the creation and usage of user defined profiles, based on the MOF standard. The consistency of models is enforced by well-formedness rules that are coded in the meta-models using OCL-like constructs. The *UML14 profile* is currently the default profile in SMW. It has been implemented to support the definitions of the UML 1.4 standard, where model elements (classes, diagrams, associations, etc) have been described. More information on the UML profile in SMW can be found in [62]. In addition, a *SART profile* was built to provide support for the SA/RT meta-model [47]. The profile is a MOF-based extension of the SMW tool that allows to graphically create, edit and manipulate SA/RT models.

The SMW tool along with its UML14 and SA/RT profiles were used to provide support for our model-driven approach. Following, we show how we have implemented the model transformations between different views of the system, using the scripting and modeling facilities of SMW.

### 5.2 Model transformations

A model transformation maps a source model expressed in a given language into a target model expressed in the same or different language by applying a number of elementary operations over elements of the model. During a model transformation we perform two types of elementary operations: *queries* and *element transformations*. These operations are implemented using scripts. Queries are scripts used to gather information (e.g., a collection of elements) from a given model that meets a given condition, or certain metrics that provide design quality information about the model. Element transformations operate over model elements by creating, modifying and erasing them.

The main difference between queries and element transformations is that queries are free of side-effects; they only provide information on a given model, but do not modify the model in any way. Usually, when performing transformations between models, one has to select the source elements using queries and then to

transform them into target elements by applying several element transformations.

We have used model transformations to provide support for the steps of the design flow presented in Sect. 4. Due to space reasons, we only present two of the model transformations supporting the design flow in Fig. 6, namely steps (d) (transforming a use case diagram into an initial object diagram) and (i) (transforming a data-flow diagram into an object diagram). The rationale behind these transformations and the corresponding algorithms were presented in the previous sections; here we only intend to describe the practical aspects involved in implementing such transformations.

### *5.2.1 From use case diagram to initial object diagram*

As presented in Sect. 4.2 the transformation is based on the 4SRS method. The script supporting this transformation provides an example of a model transformation between models of the same meta-model (i.e., UML). We start by loading the source model (use case diagram – *UseCases*) and create a target model (the *InitialObjectDiagram*).

```
1 ucModel=io.loadModel("UseCases.smw")
2 oModel=Model(name="InitialObjectDiagram")
3 elementMap={}
```

The last line of the code above, initializes a Python dictionary to keep track of how elements of the source model map into elements of the target model. Basically, a Python *dictionary* is a collection of elements indexed by a unique "key" element.

Once the source model is loaded, all model elements of type `Actor` are collected using a OCL-like query (lines 4, 5). For each identified element, a new Actor is created (line 7) and added to the target model (line 8). The pair of actors is saved in the `elementMap` dictionary (line 9).

```
4 ucActors=ucModel.ownedElement.select(lambda x:
5 x.oclIsKindOf(Actor))
6 for act in ucActors:
7 p=Actor(name=act.name)
8 targetModel.ownedElement.append(p)
9 elementMap[act]=p
```

Similarly, we collect all the use case elements in the source model (lines 10, 11) and, for each element found, three corresponding classes are created (lines 13, 19, 24) and added to the target model (lines 29–31). Each class bears the name of the use case that generated it, a stereotype specifying its category (lines 14, 20, 25) and a tag (lines 15–17, 21–23, 26–28) based on the tag name of the initial use case. For instance, the use case "{1.} Forward Datagram" (Fig. 7) will generate a ≪`control`≫ class labeled "{1.c} Forward Datagram" (Fig. 8).

```
10 useCases=ucModel.ownedElement.select(lambda x:
11 x.oclIsKindOf(UseCase))
12 for el in useCases:
13 p1=Class(name=el.name)
14 p1.stereotype.append(Stereotype(name = "interface"))
15 p1.taggedValue.append(UML14.TaggedValue(
16 name=el.taggedValue[0].name+".i",
17 dataValue=None,modelElement=p1,type=td))
18 elementMap[el]=p1
19 p2=Class(name=el.name)
20 p2.stereotype.append(Stereotype(name = "control"))
21 p2.taggedValue.append(UML14.TaggedValue(
22 name=el.taggedValue[0].name+".c",
23 dataValue=None,modelElement=p1,type=td))
24 p3=Class(name=el.name)
25 p3.stereotype.append(Stereotype(name = "data"))
26 p3.taggedValue.append(UML14.TaggedValue(
27 name=el.taggedValue[0].name+".d",
28 dataValue=None,modelElement=p1,type=td))
29 targetModel.ownedElement.append(p1)
30 targetModel.ownedElement.append(p2)
31 targetModel.ownedElement.append(p3)
```

Once the corresponding classes for each use case are created, associations are added between the ≪`interface`≫ and ≪`control`≫, and the ≪`control`≫ and ≪`data`≫ objects, respectively (lines 32–33).

```
32 a1=addAssoc(p1,p2,"")
33 a2=assAssoc(p2,p3,"")
```

The mechanism for adding an association between two classes is presented in function `addAssoc` below. The function receives as arguments two class identifiers from the target model and the corresponding association in the source model, and returns a new association element between the two classes in the target model. Initially, a new UML `Association` element is created and added to the target model (lines 35, 36). In the UML1.4 metamodel, the element that links an `Association` to a model element is `AssociationEnd`. The relationship between `Association`, `AssociationEnd` and `Class` elements is that an `Association` has two `AssociationEnd` elements corresponding to its endings. Each `AssociationEnd` element is also linked to a corresponding `Class` connected to the association. To add an association (i.e., `as1`) between two previously created classes (i.e., `c1` and `c2`), two new `AssociationEnd` elements `ase1` and `ase2` are created (lines 37–40). They are linked to the `p1` and `p2` classes through the `participant` property of the `Association` element and they are set to belong to `as1`. Consequently, the `ase1` and `ase2` elements are added (lines 41, 42) to the `association` property of the connected classes `c1` and `c2`, respectively. Finally, the newly created association is stored in the `elementMap` dictionary (line 40).

```
34 def addAssoc(c1, c2, as):
35 as1=Association(name=as.aname)
36 targetModel.ownedElement.append(as1)
37 ase1=AssociationEnd(participant=c1,
38 association=as1,multiplicity=None,isNavigable=1)
39 ase2=AssociationEnd(participant=c2,
40 association=as1,multiplicity=None,isNavigable=1)
41 c1.association.append(ase1)
42 c2.association.append(ase2)
43 elementMap[as]=as1
44 return as1
```

In the final step, the transformation script adds associations between actors and ≪interface≫ classes corresponding to the associations Actors-UseCase in the source model. Thus, all Actor-UseCase associations in the use case diagram are selected (lines 45, 46) and for each such an association the elements (i.e., Actor-UseCase pairs) linked by the given association (lines 47–53) are selected. We invoke the `addAssoc` function presented above to add associations between the corresponding pairs of elements in the target model, using the mapping information stored in the `elementMap` dictionary (line 54).

```
45 ucAssoc=ucModel.ownedElement.select(lambda x:
46 x.oclIsKindOf(Association))
47 ucAssoc.select(lambda as:
48 ucModel.ownedElement.select(lambda x:
49 x.oclIsKindOf(Actor) and
50 as.connection[0] in x.association and
51 ucModel.ownedElement.select(lambda y:
52 y.oclIsKindOf(UseCase) and
53 as.connection[1] in y.association and
54 addAssoc(elementMap[x], elementMap[y],as))))
```

The avid reader may notice our use of OCL-like constructs in an imperative manner. This was allowed by the way `lambda` functions are implemented in Python, and it greatly improved the flexibility and the level of abstraction of the scripts.

Once the transformation is complete, the SMW Toolkit allows us to save the `targetModel` into a file or to export it in the XMI format, providing the possibility to import the generated model in other UML tools.

### *5.2.2 From Data-flow diagram to Object Diagram*

In the second example, we present parts of the model transformation script implementing the step i. of Fig. 6 (i.e., transforming a DFD into a object diagram). The script provides an example of a model transformation between a source model, expressed in given language/meta-model (i.e., DFD), and a target model expressed in a different language/meta-model (i.e., UML). The algorithm and rationale of this transformation have been presented in Sect. 4.5.1.

Initially, the transformation script loads the source model and creates a new target UML model (lines 1, 2).

```
1 dfdModel=io.loadModel("dfdInput.smw")
2 targetModel=UML14.Model(name=dfdModel.name)
```

Then, the top-level data transformation of the DFD model (i.e., `topDfd`) is identified (lines 3, 4), and model (element) information from its lower-level DFD is gathered by performing a number of OCL-like queries (lines 5–13). The low-level DFD, which according to the SA/RT meta-model, is a refinement of the top-level DFD, contains the data-flow diagram on which we focus our example. The diagram was presented in Fig. 9.

```
3 topDfd=dfdModel.ownedElement.select(lambda x:
4 x.oclIsKindOf(DataTransformation))[0]
5 ee=dfdModel.ownedElement.select(lambda x:
6 x.oclIsKindOf(ExternalEntity))
7 dt=topDfd.ownedElement.select(lambda x:
8 x.oclIsKindOf(DataTransformation))
9 df=topDfd.ownedElement.select(lambda x:
10 x.oclIsKindOf(DataFlow) and
11 not x.oclIsKindOf(DataStore))
12 ds=topDfd.ownedElement.select(lambda x:
13 x.oclIsKindOf(DataStore))
```

In the following step, the script transforms each `External Entity` in the DFD into a UML `Actor` in the UML model, verifying that an actor cannot be added to the class diagram more than once (lines 9–15). As in the previous example, we use the `elementMap` dictionary to store pairs of source-target elements of the two models (line 19).

```
14 elementMap={}
15 for e in ee:
16 if e not in elementMap:
17 act=Actor(name=e.name)
18 targetModel.ownedElement.append(act)
19 elementMap[e]=act
```

For each data transformation in the dfdModel a new object (class) element is added to the `targetModel`. Firstly, pairs of `DataFlow` or `DataTransformation` elements are selected from the source model and a corresponding `Class` is added to the `targetModel` (lines 26–29). Adding a new class to the UML model is implemented by the function `addClass`, that creates a class with a given name and adds it to the model. We use again the `elementMap` dictionary to store the correspondence between the source and target elements (line 24).

```
20 def addClass(initialElement, className):
21 if initialElement not in elementMap:
22 newClass=Class(name=className)
23 targetModel.ownedElement.append(newClass)
24 elementMap[initialElement]=newClass
25 return newClass
26 topDfd.ownedElement.select(lambda ts:
27 (ts.oclIsKindOf(DataTransformation) or
28 ts.oclIsKindOf(DataStore)) and
29 addClass(ts, ts.name))
```

Again, one can notice, in line 29, the use of OCL-like constructions in an imperative manner.

Once the corresponding number of classes is added to the `targetModel`, the script creates the associations among them. As presented in Sect. 4.5.1 we have three different cases of associations between elements based on the source and target elements of the data flows in the DFD model. The first case is that of an initial data flow linking two data transformations. For each pair of source-target data transformations, their corresponding classes are identified in the `targetModel` and a `send()` association is added (lines 30–39), using the function `addAssoc()` presented in the previous example.

```
30 topDfd.ownedElement.select(lambda f:
31 f.oclIsKindOf(DataFlow) and
32 topDfd.ownedElement.select(lambda src:
33 src.oclIsKindOf(DataTransformation) and
34 f.connection[0] in src.association and
35 topDfd.ownedElement.select(lambda dst:
36 dst.oclIsKindOf(DataTransformation) and
37 f.connection[1] in dst.association and
38 addAssoc(elementMap[src],elementMap[dst],
39 "send"+string.split(f.name,'+')[0]))))
```

In the second case, we treat data flows that either originate from or have as target a data store element in the `dfdModel`. This type of data flows will be transformed into `read()` or `write()` associations, depending on the direction of the initial data flow.

The third case of associations is based on the data flows that communicate with external entities of the DFD model. This type of data flows will generate `set()` associations. Since in the second and third case the code is similar to the code presented in lines 30–39, we intentionally omit it here.

Finally, for those classed linked by associations, one of the classes receives methods and encapsulated attributes corresponding to the name of the associations. The `addAssocMeth` function below is used to add a new operation (method) and a corresponding attribute to a given class (lines 40–47). In this sense, classes originating from data stores receive as attributes the parameters of the data flows and corresponding methods to read and/or write those parameters (lines 48–51).

```
40 def addAssocMeth(clas, methName, prefix):
41 o=Operation(name=methName)
42 if methName not in clas.feature.name:
43 clas.feature.insert(o)#
44 attr=Attribute()
45 attr.name="someName"
46 clas.feature.insert(attr)
47 return 1
48 ds.select(lambda t: df.select(lambda f:
49 f.connection[1] in t.association and
50 addAssocMeth(elementMap[t],
51 string.split(f.name,'+')[0],"write")))
```

In both examples, we intentionally omitted initialization and other code lines not relevant to the transformation process. Also, in both examples we have used the scripts to transform diagrams placed in a source model file into another diagram placed in a different target model file. But nothing prevents us from using the scripts to operate between the diagrams placed inside the same model, as long as the meta-model allows the coexistence of the diagrams inside the same model.

## 6 Conclusions

The combination of object-oriented and functional approaches is almost universally seen as a “bad” solution to software modeling. However, we believe that it can give useful results in some specific contexts if not seen as an infallible approach, but instead used with some caution.

In this paper, the combination of the functional and object-oriented approaches, represented by DFDs and UML, respectively was analyzed. The emphasis of the discussion was put on topics related to the analysis phase of embedded software systems. The rationale was to always have as the major model of the implementation phase, some object or class diagram so that an object-oriented programming languages could be used, but also to 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. We have also shown that it is possible to create and manipulate artifacts described by both DFD and UML notations, and also to implement automated transformations among different steps of the design process.

The automated transformations have to be viewed as an aid to the designer and not something to replace him/her. Since the models of both UML and DFD have a user-friendly view, using the SMW tool enables support for human (i.e., designer) intervention. The way the diagrams were created and transformed allows the designer to trace to what pieces of functionality different elements in the system belong. The above transformations are in some situations reversible allowing the designer to repeatedly change the view during the system specification and design. At each change of view, new details specific to one of the views can be added until the necessary level of detail is reached.

The presented model-driven engineering approach promotes a mapping between structural (object diagram) and dynamic (DFD) models. If a proper balance between these models is achieved, the main advantage of the approach is that the benefits of both models, in

terms of expressiveness and focus, apply simultaneously. However, if both models are biased towards one of the perspectives, we actually have two different diagrams for the same purpose, one of them being useless.

For the moment, since we were addressing protocol processing applications targeted to hardware platform implementations, we intentionally avoided referring to specific object-oriented mechanisms as inheritance and polymorphism. Their dynamic nature is in contrast with the static nature of the hardware components.

The models presented here could be obtained from scratch, following some of the techniques proposed by several object-oriented methods, or as the result of transforming the previous DFD into object/class diagrams. In this last situation, we must state that the transformation of DFDs into an object model is not at all straightforward. Usually, transforming requirements into a software architecture (i.e., the transition from analysis to design) is not easy and here there is an additional difficulty, that results from the paradigm shift. Both models obtained from the DFD diagram were also used for developing prototypes in Java, in order to demonstrate their adequateness to describe the system. The prototypes were built with the idea of showing that the models do constitute a valid solution for the implementation of the system under consideration. The Java program code is not included here, but can be downloaded from [63].

As a final note, it intrigues us why the adoption of use cases, within the context of object-oriented development, is so popular and considered a suitable technique in object-orien-ted design, given that DFDs also produce a functional decomposition of a system. The answer could lie 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 case diagrams. Use case modeling is quite useful when the development team needs to discuss the requirements of a system with its stakeholders, especially users, managers, customers and clients. This occurs because use case diagrams are an easy-to-read notation and, due to their simplicity and to the intuition behind the *use case* and *actor* concepts, promote the participation of non-technical stakeholders. These characteristics are 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. DFDs are good for systems that present these characteristics.

Taken into 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 it may not be easy or simple. If this is accepted to be suitable from the use case diagrams (in conjunction with other models, such as sequence and collaboration diagrams), the same thing should also be possible, and easier we ought to add, with DFDs and those same additional diagrams.

As future work, the following four 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 method is also a future path for continuing this work. Finally, it is important to devise a more rigorous method for data classification inside DFDs, to bring out the benefits of object-oriented mechanisms like inheritance and polymorphism. For instance, we can easily see that the `routingDatagram` class in Fig. 14 looks similar to a parent class of the `responseDatagram` and `requestDatagram`.

**Acknowledgements** Financial support for J. M. Fernandes from CIMO (grant HH-02-383) and from FCT and FEDER under project METHODES (POSI/37334/CHS/2001) and for D. Truscan from the HPY and TES research foundations is gratefully acknowledged.

## References

1. Fernandes, J.M., Lilius, J.: Functional and object-oriented modeling of embedded software. In: 11th International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'04) (2004)
2. Truscan, D., Fernandes, J.M., Lilius, J.: Tool support for DFD-UML model-based transformations. In: 11th International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'04) (2004)
3. Hatley, D.J., Pirbhai, I.A.: Strategies for Real-Time System Specification. Dorset House (1987)
4. Ward, P.T., Mellor, S.J.: Structured Development for Real-Time Systems. Prentice Hall/Yourdon Press, Englewood Cliffs (1985) (Published in 3 volumes)

5. Dieste, O., Genero, M., Juristo, N., Maté, J., Moreno, A.: A conceptual model completely independent of the implementation paradigm. J Syst Softw **68**(3), 183–198 (2003)
6. Machado, R.J. Fernandes, J.M., Monteiro, P., Rodrigues, H.: Transformation of UML Models for Service-Oriented Software Architectures. In: Proceedings of 12th IEEE international conference on the engineering of computer based systems (ECBS 2005), pp. 173–82 (2005)
7. Ambler, S.W.: Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. Wiley, New York (2002)
8. Alabiso, B.: Transformation of data flow analysis models to object oriented design. In: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88), pp. 335–353. ACM Press, New York (1988)
9. Shoval, P., Kabeli, J.: FOOM: functional- and object-oriented analysis and design of information systems – an integrated methodology. J Database Manage **12**(1), 15–25 (2001)
10. Ward, P.T.: How to integrate object orientation with structured analysis and design. IEEE Softw **6**(2), 74–82 (1989)
11. Periyasamy, K., Mathew, C.: Mapping a functional specification to an object-oriented specification in software re-engineering. In: 24th ACM Annual Conference on Computer Science (CSC '96), pp. 24–33. ACM Press, New York (1996)
12. de Champeaux, D. et al. Panel: structured analysis and object oriented analysis. In: ECOOP/OOPSLA, pp. 135–139. ACM Press, New York (1990)
13. Sodan, A.C.: Yin and Yang in computer science. Commun ACM **41**(4), 103–111 (1998)
14. Kim, J., Ferch, F.J.: 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. 489–498. ACM Press, New York (1992)
15. Glass, R.L.: The naturalness of object orientation: beating a dead horse? IEEE Softw **19**(3), 103–104 (2002)
16. Vessey, I., Conger, S.A.: Requirements specification: learning object, process, and data methodologies. Commun ACM **37**(5), 102–13 (1994)
17. Jackson, M.: Software requirements and specifications: a lexicon of practice, principles and prejudices. ACM Press, New York (1995)
18. Hatley, D.J., Hruschka, P., Pirbhai, I.A.: Process for System Architecture and Requirements Engineering. Dorset House (2000)
19. Douglass, B.P., Harel, D., Trakhtenbrot, M.: Statecharts in use: structured analysis and object-orientation. In: Lectures on Embedded Systems, LNCS 1494, pp 368–394. Springer, Berlin Heidelberg New York (1998)
20. Chen, P.: Entity-relationship modeling: historical events, future trends, and lessons learned. In: Software Pioneers: Contributions to Software Engineering, pp. 297–310. Springer, Berlin Heidelberg New York (2002)
21. Swartout, W., Balzer, R.: On the Inevitable Intertwining of Specification and Implementation. Commun ACM **25**(7), 438–440 (1982)
22. Girault, A., Lee, B., Lee, E.A.: Hierarchical finite state machines with multiple concurrency models. IEEE Trans Comput Aid Des of Integ Circuits Syst **18**(6), 742–760 (1999)
23. Howerton, W.G., Hinchey, M.G.: Using the right tool for the job. In: 6th IEEE International Conference on Complex Computer Systems (ICECCS '00), pp. 105–115. IEEE CS Press (2000)
24. Lee, E.A.: Computing for embedded systems. In: 18th IEEE Instrumentation and Measurement Technology Conference (IMTC/2001) (2001)
25. Dori, D.: Object-Process Methodology – A Holistic Systems Paradigm. Springer, Berlin Heidelberg New York (2002)
26. Peleg, M., Dori, D.: Extending the object-process methodology to handle real-time systems. J Object Orient Program **11**(8), 53–58 (1999)
27. Wang, E.Y., Cheng, B.H.C.: Formalizing and integrating the functional model into object-oriented design. In: Proceedings of SEKE '98 (1998)
28. Becker, L.B., Pereira, C.E., Dias, O.P., Teixeira, I.M., Teixeira, J.P.: 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 (2000)
29. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall International, Englewood Cliffs (1991)
30. Gall, H., Klösch, R.: Finding objects in procedural programs: an alternative approach. In: 2nd Working Conference on Reverse Engineering, pp. 208–216. IEEE CS Press (1995)
31. Jacobson, I., Lindström, F.: Reengineering of old systems to an object-oriented architecture. In: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '91), pp. 340–350. ACM Press New York (1991)
32. Pnueli, A.: Embedded systems: challenges in specification and verification. In: Sangiovanni-Vincentelli, A., Sifakis, J. (eds.) Embedded Software, Second International Conference EMSOFT 2002, LNCS 2491, pp. 1–14. Springer, Berlin Heidelberg New York (2002)
33. Lilius, J., Truscan, D.: UML-driven TTA-based protocol processor design. In: Forum on specification and design languages (FDL '02) (2002)
34. Ambler, S.W.: What's Missing from the UML? SIGS Publications, Object Magazine (1997)
35. Fernandes, J.M., Duarte, F.J.: A reference framework for process-oriented software development organizations. Software and Systems Modeling. Springer, Berlin Heidelberg New York (2004) http://dx.doi.org/10.1007/s10270-004-0063-0.
36. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley, Reading (2002)
37. Sigfried, S.: Understanding Object-Oriented Software Engineering. IEEE Press (1996)
38. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs (1988)
39. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Reading (1998)
40. Fernandes, J.M., Machado, R.J.: From use cases to objects: an industrial information systems case study analysis. In: 7th International Conference on Object-Oriented Information Systems (OOIS '01), pp. 319–328. Springer, Berlin Heidelberg New York (2001)
41. Lyons, A.: UML for real-time overview. Technical report, ObjecTime Limited (1998)
42. Selic, B.: Turning clockwise: using uml in the real-time domain. Commun ACM **42**(10), 46–54 (1999)
43. Martin, G., Lavagno, L., Louis-Guerin, J.: Embedded UML: a merger of real-time uml and co-design. In: 9th ACM/IEEE/IFIP International Symposium on Hardware/Software Codesign (CODES '01), pp. 23–28. ACM Press New York (2001)
44. Lee, E.A., Parks, T.M.: Dataflow process networks. Proc IEEE **83**(5), 773–801 (1995)
45. Wolf, W.: Computers as Components: Principles of Embedded Computing System Design. Morgan Kaufman Publishers (2000)
46. OMG. OMG Meta-Object Facility (MOF). Document formal/01-11-02, http://www.omg.org.

47. Isaksson, J., Lilius, J., Truscan, D.: A MOF-based metamodel for SA/RT. In: Proceedings of Rapid Integration of Software Engineering techniques (RISE'04) workshop. Luxembourg, Luxembourg, 26 November 2004. LNCS 3475, pp. 102-111, Springer, Berlin Heidelberg New York (2005)
48. Jacobson, I.: Basic use case modeling (continued). Report on Object Anal Design **1**(3), 7–9 (1994)
49. Harel, D., Rumpe, B.: Modeling languages: syntax, semantics and all that stuff – part i: the basic stuff. In: Technical report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel (2000)
50. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: ECOOP '97 – Object-oriented programming, LNCS 1241, pp. 140–149. Springer, Berlin Heidelberg New York (1997)
51. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley, New York (1994)
52. Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading (1992)
53. Rosenberg, D., Scott, K.: Use Case Driven Object Modeling with UML: A Practical Approach. Addison-Wesley, Reading (1999)
54. Ying, L.: From use cases to classes: a way of building object model with UML. Inform Softw Technol **45**(2), 83–93 (2003)
55. Pawson, R.: Naked objects. IEEE Softw **19**(4), 81–83 (2002)
56. OMG. Unified Modeling Language Specification. In; Technical report, OMG (2002)
57. OMG. OMG model driven architecture, July 2001. Document ormsc/2001-07-01. http://www.omg.org.
58. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software developement. IEEE Softw **20**(5), 42–45 (2003)
59. Truscan, D., Fernandes, J.M., Lilius, J.: Tool support for DFD to UML model-based transformations. In: Technical report 519, TUCS, Turku, Finland (2003)
60. Open SystemC Initiative. http://www.systemc.org.
61. http://www.python.org.
62. Porres, I.: A Toolkit for Manipulating UML Models. Software and Systems Modeling. Springer, Berlin Heidelberg New York **2**(4), 262–277 (2003)
63. http://www.abo.fi/ dtruscan/ipv6index.html


## Author Biography

**João M. Fernandes** is an assistant professor at the Department of Informatics, Universidade do Minho (Braga, Portugal). He received a Lic. degree in Informatics and Systems Engineering in 1991, a M.Sc. degree in Computer Science in 1994, and a Ph.D. degree in Computer Engineering in 2000, all from Universidade do Minho. His Ph.D. thesis, entitled "An Object-Oriented Methodology for Embedded Systems Development", addresses the usage of object-oriented concepts, namely UML, to analyze, design, implement, and test embedded systems. From Sep/2002 until Feb/2003, he was a post-doctoral researcher at the TUCS Embedded Systems Laboratory (Turku, Finland). His research interests focus on Embedded Software, Hardware/Software Co-Design, Methodologies for System Development, Software Modelling, Software Process and Management, and History of Computing. For more information, consult his website at http://www.di.uminho.pt/~jmf.

**Johan Lilius** got his MSc (Tech.) from Helsinki University of Technology, Finland, in 1989, and his DSc Tech from Helsinki University of Technology, Finland in 1995. Since 2001, he has been a Professor in Computer Engineering at Åbo Akademi University, Finland, where he also heads the Embedded Systems Laboratory. His research interests include UML and model-driven architecture, hardware/software codesign, model-checking and low-power design for software.

**Dragos Truscan** is a Ph.D student with the Embedded Systems Laboratory, Turku Centre for Computer Science, Finland. He got his Engineer Diploma in Computer Science from the Department of Computer Science and Engineering, University "Politehnica" of Bucharest, Romania, in 1999. His research interests include model-driven engineering, hardware/software codesign, high-level specification of embedded systems, protocol processors design, transport-triggered architectures.