# Tool Support for DFD-UML Model-based Transformations

Dragos Truscan João M. Fernandes * Johan Lilius
*Embedded Systems Laboratory*
*Turku Centre for Computer Science and Åbo Akademi*
*Lemminkäisenkatu 14A, 20520 Turku, Finland*
*e-mail: dragos.truscan@abo.fi; jmf@di.uminho.pt; johan.lilius@abo.fi*

## Abstract

*This paper presents a model-based approach that combines the data-flow and object-oriented computing paradigms to model embedded systems. The rationale behind the approach is that both views are important for modelling purposes in embedded systems environments, and thus a combined and integrated usage is not only useful, but also fundamental for developing complex systems. We also show that by using models we were able to implement automated transformations between different views of the system under design. We exemplify the approach with an IPv6 router case study.*

## 1. Introduction

The increasing complexity of today's embedded systems requires new approaches to be applied to their specification and design. On one hand there is a need to raise the level of abstraction to the point where the designer focuses more on the concepts and different views of the system and less on the implementation details. On the other hand, the approach should provide the necessary support for automation, consistency checking and verification, thus reducing the development costs and time of the new systems. In order to tackle these problems, one should use models to describe systems, starting from the high-level specification until the platform implementation.

To better exploit the possibility of automation provided by models, appropriate tool support is needed, allowing the designer to navigate between different views of the system in a tool supported manner. Recently, OMG started to promote a new vision to system development: Model Driven Architecture [18], 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 motivation of MDA is to move focus from platform implementation to solution (business) modelling.

In the last couple of years object-orientation have constantly gained support and the majority of the software modelling techniques has adopted it. This is probably due to the observable emphasis on data in system design that the object-oriented paradigms provide [16]. But as pointed out in several articles [8, 11], one conceptual model (e.g. object-oriented methods) 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, a designer should be able to switch from one view to another at different levels of abstraction during the design process.

Beside object-orientated methods, several other languages have been proposed and used for similar purposes (i.e. development of embedded systems). One such language is provided by the structured-analysis methods introduced in early 70's, which became quite popular in industrial environments. In recent years, the structured-analysis methods were largely overshadowed by the object-oriented design methods, especially after the introduction of UML. Although nowadays 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. We consider that both object-oriented and structural analysis methods represent viable and necessary tools in embedded systems' design, each of them providing important techniques for describing the system under consideration.

One of the aims of this work is to study the integration of data-flow diagrams (DFDs), the main tool of the structured analysis methods, with object-oriented diagrams provided by UML, using a model-based approach. The main motivation for this comes from a case study [17], where we have applied object-oriented techniques and UML to the design of a protocol processor. In this case study we had

*On leave from Dept. Informatics, Universidade do Minho, 4710-057 Braga, Portugal.

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

IEEE COMPUTER SOCIETY

in many situations the feeling that there was a functional and structural view of the system that was not adequately represented by the diagrams provided in UML. A second motivator for this work has been the experience of the 3rd author, that at least in the Turku, Finland region, the embedded systems industry is reluctant to move from structured to object-oriented methods. There is therefore a need for studying the interrelation between structured and object oriented methods. Third but not less important, industry needs nowadays to cope with high levels of design complexity, that can be tackled using model-based approaches to provide automation and consistency checking during the analysis and design phases of embedded systems.

The main contributions of this paper are: (a) to define a model-based approach that integrates data-flow and object-oriented views for specification of embedded systems, (b) to identify a set of model transformations for moving between the data flow model and the object-oriented one and (c) to provide tool support for these transformations using automated scripts. We will use a simplified IPv6 router case study to illustrate the approach.

Following we present, a brief overview of the related work on DFD and object-oriented integration (Section 2), then we will briefly introduce, in Section 3, the SMW tool that we used to implement the two models and automate the transformations between the views. Section 4 will present the main steps of our design flow, where we show how we perform transformations between structured and object-oriented views of the specification at system-level, and the main steps to be taken in order to provide an automatable approach. In Section 5 a short transformation example is presented. The paper ends with some conclusions and comments about these transformations.

## 2. Related work

Many authors have already studied the combination of DFDs with object-oriented methods. A theoretic survey of this topic is given in [10]. 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 [9]. 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 [19].

In [32], 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, while 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 [6], the functionality associated with each use case is described as an E-DFD (an extended version of the traditional DFD). A tool, called SysObj, uses these inputs to automatically generate an object model, that is viewed as the architecture of the system. This method is also related to an integrated environment for developing distributed systems with the object-oriented paradigm [5].

In OMT, DFDs are also used to describe the functional model of a system [22]. 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. 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 [12]. 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 [14].

Alabiso also proposes the transformation of DFDs into objects [3]. 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) [24], which is specifically tailored for information systems. The main idea behind FOOM 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 the FOOM, the specification of user requirements is accomplished, in functional terms, by OO-DFDs (a DFD with data stores re-


0-7695-2125-8/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

placed by classes), and in data terms by an initial object-oriented schema, or an ERD which is easily transformed into an initial object-oriented schema. In the design phase, the artifacts from the analysis are used and a detailed object-oriented and a behavior schemas 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 at hand.

## 3. The SMW Tool

To create and manipulate the UML and DFD models, we make use of the freely available Software Modelling Workbench (SMW) tool available for download at URL `www.abo.fi/~iporres/html/smw.html`. The tool is built upon the OMG's MOF and UML standards, allowing edition, storage and manipulation of metamodels. SMW uses the Python language to describe the elements of a metamodel, each element being represented by a Python class. This fact provides the basic scripting mechanisms for querying and manipulating given models. Moreover, making use of the *lambda*-functions that the Python language provides, OCL-like idioms are supported. OCL (Object Constraint Language) is a semi-formal language, developed by OMG, to add more precision and less ambiguity to the UML metamodels, beyond the capabilities of the graphical diagrams. Usually, OCL is used as a declarative language, to interrogate parts of a UML specification, thus being free of side effects. More information on the UML profile and SMW can be found in [21].

SMW allows the creation and usage of user defined profiles, based on the MOF standard. The Python implementation of a given metamodel can be obtained from either a DTD file or by creating a UML class diagram saved in the XMI format. In addition, the consistency of models is provided, well-formedness rules being coded into Python using OCL-like constructs. To implement the UML and DFD paradigm we use the UML14 and SA/RT SMW profiles.

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 among them, etc) have been described. In addition, the constraints and requirements of UML are enforced using OCL. The **SA/RT profile** has been built in the SMW tool to implement the SA/RT metamodel [13]. Structured Analysis for Real-Time Systems, or SA/RT, is a graphical design notation focusing on analyzing the functional behavior of an information flow through a system. Data Flow Diagrams (DFD) are the main diagram type used for structured analysis and for representing data-flows in the system. The profile is a MOF-based extension of the SMW tool that allows to graphically operate in the SMW over SA/RT specifications, thus benefiting from the scripting facilities of the tool.

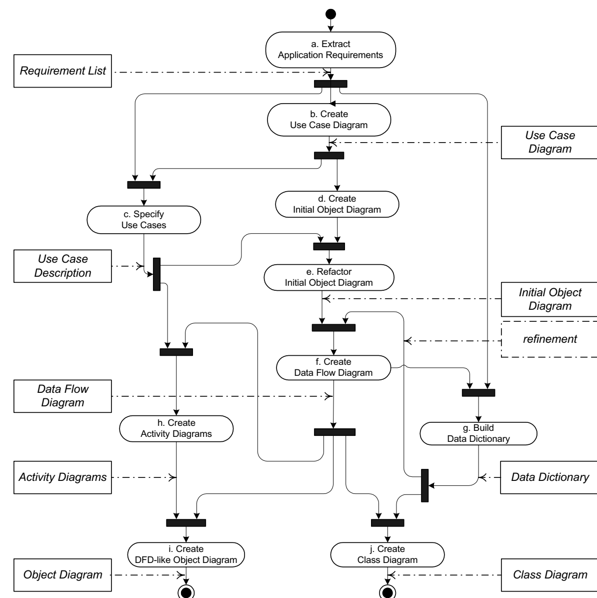


**Figure 1. Integration of UML and DFD models**

## 4. A model-based approach

We see the integration of object-oriented and DFDs from a modelling perspective, where tool support has to be provided for obtaining a real model-driven approach for the analysis and design of embedded systems. The approach consists of specifying 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 [17], but with a different perspective. There, the main objective was to define a complete UML-based design flow (or methodology) for embedded systems, making special emphasis on the real implementation of the system. Here our principal aim is to discuss the possibilities of merging DFDs with other object models, during the analysis phase activities and to create a viable tool support for creating and maintaining these models during the entire life-cycle of the design.

We make use of both UML14 and SA/RT profiles provided in SMW to be able to model the system under consideration, combining the object-oriented and data flow views for the analysis and design of embedded systems. The main phases of the process (Fig. 1) 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 IOD
e. Refactor IOD by grouping, splitting and discarding objects based on their functionality



0-7695-2125-8/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

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 automate the transition between these views, model transformations are required. Several authors recognize the model transformation as the fundamental mechanism for model-driven development [23]. A model transformation takes a source model expressed in a given language (e.g. UML) and transforms it into a target model expressed either in the same language (e.g. UML) or in a different one (e.g. DFD). We consider that model transformation is an important technique for applying software patterns and refactorings, and to provide a good support for reuse. In order to provide an automated approach, we make use of scripts to implement the transformations. A script is a software application used to interrogate, modify or create new artifacts in a model. We consider that there are 3 types of scripts in a model-driven approach: queries, model transformations and code generation scripts [4].

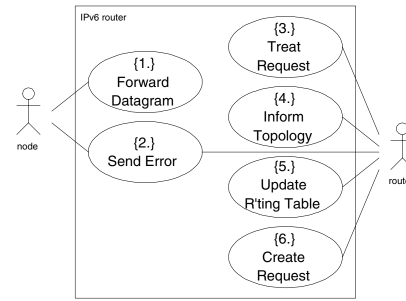


**Figure 2. IPv6 Router Use Case Diagram**

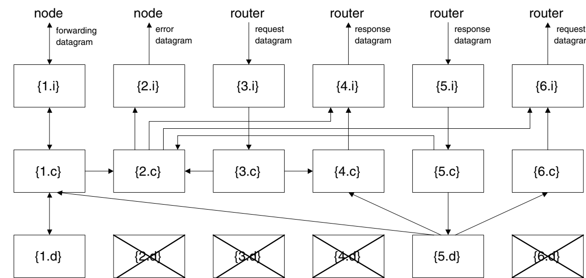


**Figure 3. Initial Object Diagram**

## 4.1. Capturing the requirements into Use Cases

We start by analyzing the specification of the router requirements and building a Use Case Diagram (Fig. 2). We identify two external actors that interact with the router. The Node is a common network node that requests the router to forward datagrams and eventually to send back an ICMPv6 error message 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 6 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 [26].

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 modelling and can be applied to any system, developed either with a structured or object-oriented approach. Indeed, following we show how we obtain a data flow diagram starting from a use cases diagram.

## 4.2. From Use Cases to Initial Object Diagram

From the Use Case Diagram 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 approach presented in [11]. The objects have the same number as the initial use case, but each of them has the appropriate tag name. For instance, the {1.i}, {1.c} and {1.d} objects (Fig. 3) represent the interface, control and data objects obtained from splitting the Forward Datagram {1.} use case. Also, a corresponding actor is created in the initial object diagram corresponding to the actors in the Use Case Diagram. Finally, we add associations between objects and also between objects and the external environment (actors).

We enforce a clear separation of the functionality that each object category has inside the system. Control objects deal only with algorithmic and control behavior. Interface objects are only placed at the border of the system to intermediate between the internal and external communication of the system. Data objects deal only with storing data and providing access methods to this data.

When a system is divided into parts, both structure and behavior are being decomposed along with functionality. Some authors [15] even consider that all systems are submitted to functional decomposition even when different decomposition paradigms are used. 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.

The main steps of transforming the Use Case Diagram


0-7695-2125-8/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

(UCD) into a an Initial Object Diagram (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 the 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 the UCD, an association is drawn between the corresponding Actor and the interface object generated by the given Use Case

### 4.3. Refactoring the Initial Object Diagram

According to the 4SRS method [11], by instantiating scenarios for the initial use cases we decide what objects are kept or disposed of, and also we identify the communication (depicted by association) among objects. The remaining set of objects are refactored, by decomposing or grouping together objects of the same type. The interface objects that are communicating the same information with the same actor and in the same direction can be grouped together (for instance {5.i} and {3.i} both receive routing datagrams, either request or response datagram, from the router). An interface object that communicates bidirectionally with one node, can be split into 2 interface objects ({1a.i} and {1b.i}), one dealing with incoming traffic and the other with the outgoing traffic. We structure the model so that all communication between data objects and interface objects is done through control objects. The final result of the refactoring is presented in Fig. 3.

Although, the splitting and grouping of objects are supported by scripts, the refactoring process is performed manually based entirely on the designer's experience. Future research will look into a more automated approach based on identification of patterns.

### 4.4. From Object Diagram To DFD

Sometimes designers need to be able to change the view they are using. We use the DFD view to be able to identify, classify and refine the data flows involved in the system.

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). They provide a data-driven view of the system useful for describing transformational systems, such as digital signal-processing systems, compilers, multimedia systems, or telecommunication devices.

We obtain the data flow model of the system from the initial set of objects (Fig. 3), by noting the direction of the communication among the objects and then defining data entities that are being exchanged. Thus, to obtain the data-flow diagram 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 involved.

The first step of the transformation process is straightforward because of the way the initial object diagram was structured. In the initial object diagram we have control objects and interface objects that are processing input communication from neighboring objects and transforming it into output communication. This is similar with the behavior of the processes provided by the DFD concept. This fact allows us to transform all the control and interface objects into DFD-specific processes (data transformation). Similarly, data objects in the initial object diagram 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 its 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 active processes would be a process that is periodically reading a data store (Fig. 4, processes {1b.c} and {6.c}). 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 in the next design phases to specify the internal behavior of each process.

To transform an IOD into a DFD the following steps must be performed:

1. transform each Actor in the IOD into an External Entity
2. transform interface and control objects into Data Transformations in the DFD
3. transform data objects into Data Stores
4. associations between elements of the IOD are transformed into Data Flows either between external entities and data transformations, or between data transformations and data stores, or in-between data transformations
5. we identify and mark active processes in the DFD

In the second step, the associations among the initial objects are transformed into input and output data flows in DFD based on the system specification. During the refinement of data flows, we focus on adding new details about the data entities and types transported over the flows. The resulting diagram is presented in Fig. 4. Also, a classification of the data flows involved in the system is performed by


0-7695-2125-8/04 $ 20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

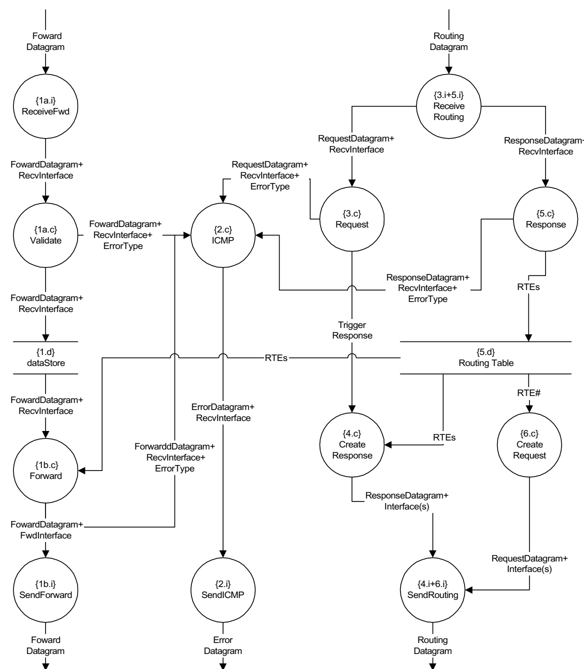


**Figure 4. Data-Flow Diagram of the Router**

building a Data Dictionary. This is done by analyzing the data that is moved between data transformations and having as primary information source the application requirements. The data flow identification is performed manually and it is based on designer's experience. A complete Data Dictionary specification of the IPv6 Router under study can be found in [10]. 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

One specific situation where the usage of DFDs is helpful is in re-engineering activities where the system was previously developed following the guidelines of some structured method. Even if the diagrams are no longer available, it is expected to be easier to reverse-engineer the program code into DFDs and other complementary models, than to transform it directly into some object-oriented models.

For obtaining a object-oriented model of the system starting from the DFD-model we have tried 2 approaches. In the first approach we transform the DFD model directly into an object diagram by transforming on an one-to-one basis the processes and data stores in the DFD model into objects. The second approach follows a more object-oriented view where we focus on classifying data in the system and detecting class methods that operate over that data.

Some similar ideas were already proposed in the FOOM methodology [24] for developing information systems, but its usage for embedded systems requires necessarily some adaptation. The transformation of a functional specification in Z into an object-oriented one in Object-Z, for re-engineering purposes, is also proposed in [20].

### 4.5.1 From DFD to object-based class diagram

In the first approach, we map the DFD model directly into an object one by transforming the artifacts in the DFD model on a one-to-one basis. Basically, the algorithm consists in transforming each data transformation in the DFD into an object in the Object Diagram, and the data flows between the data transformations into associations. In addition, the data flows involved in the system become internal attributes of the classes (objects), are encapsulated into the objects and used now to describe the internal state of the objects. In order to access the data entities inside objects, corresponding methods are added.

Objects originating from processes placed at the border of the system will have *set()* methods to communicate with the external environment, while the other processes will have *send()* methods that receive as parameters the input data flows in the DFD. 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 the 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 suggestive for the designer. Caution has to be taken that by changing names, the consistency of the design is not affected. Thus, either the modelling tool or the design method should provide support to check the consistency of the design after each transformation step.

After 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. 5) 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 reactive objects execution is triggered only when one of their methods is invoked, while the ac-


IEEE COMPUTER SOCIETY

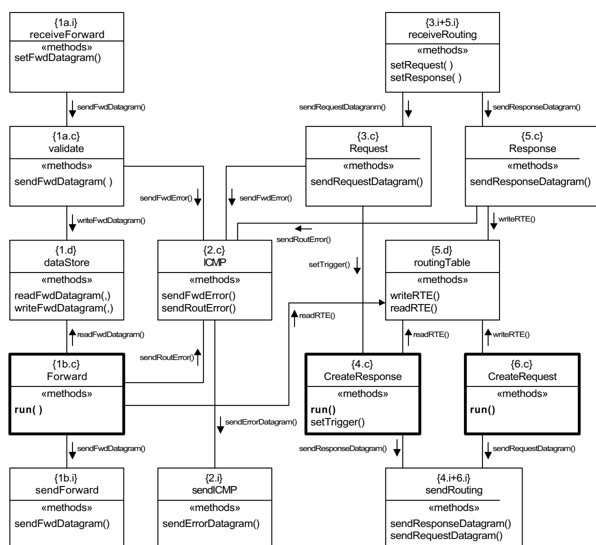


**Figure 5. The DFD-like Class Diagram**

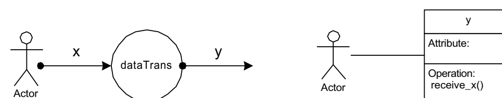


**Figure 6. Border process pattern**

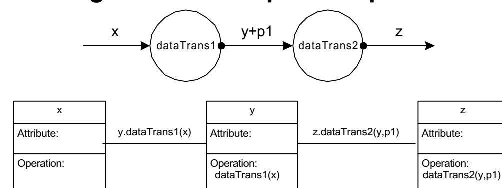


**Figure 7. Interprocess comm. pattern**

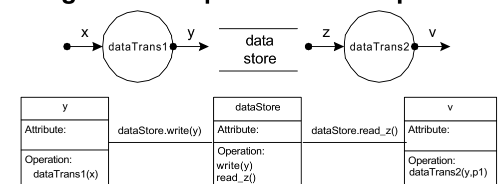


**Figure 8. Data Store communication pattern**

tive objects have a state machine (usually implemented by a `run()` method) executing continuously.

The transformation script performs the following steps:

1. each data transformation in the DFD is transformed into a new class/object in the object model
2. transform flows between transformations in DFD into class associations
3. the classes originating from processes receive input flows receive set_() method and a corresponding attribute
4. active objects receive run() method
5. objects receiving an input data-flow receive send_() method of the incoming dataflow
6. each data store is transformed into a class/object, with read and write methods, according with the input/output direction of the flows connected to the data store

The object diagram in Fig. 5 provides a low level of abstraction and data encapsulation, but it proved to be quite 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 our TACO processor platform [17]. The TACO protocol processor is based on the Transport Triggered Architecture (TTA) [7]. In TTA processors data transports are programmed and they trigger operations - traditionally operations are programmed and they trigger transports. A TTA processor is composed of functional units (FUs) 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 [2]. SystemC is an object oriented (extension of C++) hardware specification language where the hardware modules are instances of SystemC classes.

Due to the architectural aspects of the processor a DFD-like approach proved well suited for the specification and design phases. Objects in Fig. 5 naturally map to FUs and the methods of the objects have similar granularity with the operations provided by the TACO processor. To configure TACO to support a given application one has to select a number of required resources from the component library (modeled as SystemC class diagram). The selection is done by analyzing the object diagram of the of the specification and selecting from the SystemC class diagram those resource supporting operations. 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 take 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 implemented by the transformation script 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.


IEEE COMPUTER SOCIETY

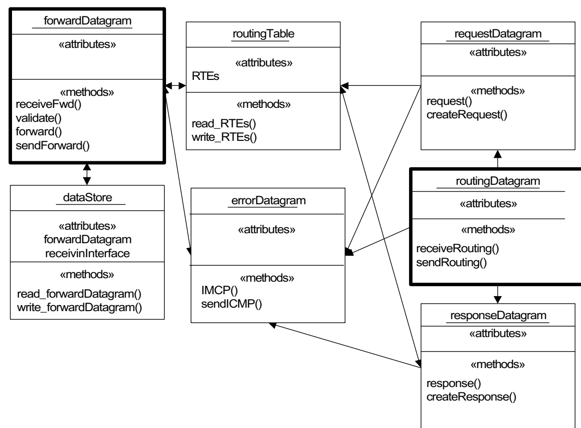


**Figure 9. The class diagram of the router**

In order to add class methods the script looks for three kinds of patterns in the data flow diagram: data flows communicating with the external environment of the system (Fig. 6), data flows between two processes (Fig. 7) and data flows that communicate with data store elements (Fig. 8).

A number of processes (data transformations) operate over each data flow class inside the DFD. For instance, the ForwardDatagram data flow (Fig. 4) is processed by different data transformations (ReceiveFwd, Validate, Forward, SendForward, ICMP). Thus, we create the forwardDatagram class inside the class diagram (Fig. 9) and we add the DFD processes that affects it as 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. 4 is not transformed into a method of forwardDatagram because it outputs an ErrorDatagram.

The 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 for accessing data based on the input and output flows of the initial data store (see `dataStore` in Fig. 8). In addition, we classify the data inside the "Data Store" classes based on the data flows that access the Data Store element.

We consider that here we also have a possible classification of classes into active and reactive. The idea behind this classification is based on the life-cycle of objects instantiated from classes. The active classes are those that instantiate objects of other classes, while the reactive classes are the ones whose objects are instantiated by others classes. One example of an active class is the forwardDatagram class (Fig. 9) because it might instantiate objects of other classes (i.e. errorDatagram). Classifying classes into active and reactive is for the moment an ad-hoc approach, but we intend to further investigate it.

The transformation script performs the following steps:

1. transform distinct data flows and data stores into classes
2. transform each external entity into an actor
3. apply the Inter-process Communication Pattern
4. apply the Border Process Pattern
5. apply the Data Store Communication Pattern

## 5. Implementing model transformations

As presented in the introduction we use the SMW tool and its UML14 and SA/RT profiles to model the object oriented and data flow views of the system. Following, we show how we were able to implement model transformations between different views of the system, by using the scripting and modeling facilities of SMW.

A model transformation transforms 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 model elements. Usually, one has to select the source elements and then to transform them into target elements by applying the model transformation.

Due to space reasons we present as example only parts of one of the transformations (step i. in Fig. 1). We gather model information (elements) form the DFD by performing a number of OCL-like queries (lines 1-8):

```
1 dataFlows=dfdModel.ownedElement.select(lambda x:
2    x.oclIsKindOf(DataFlow))
3 externalEntities=topDfd.ownedElement.select(lambda x:
4    x.oclIsKindOf(ExternalEntity))
5 dataStores=topDfd.ownedElement.select(lambda x:
6    x.oclIsKindOf(DataStore))
7 dataTransformations=topDfd.ownedElement.select(lambda x:
8    x.oclIsKindOf(DataTransformation))
```

Next, we transform each external entity in the DFD into a UML actor in the OD, verifying that an actor cannot be added to the OD more than once (lines 9-15).

```
9  dfdModel.ownedElement.select(lambda act:
10     act.oclIsKindOf(ExternalEntity) and
11     (s.refinedFlow.connection[0] in act.association
12     or s.refinedFlow.connection[1]))
13 if topAct.name[0] not in actors:
14     newActCl=classDiag.addActor(name=topAct.name[0])
15     actors.append(newActCl.name)
```

Then for each data transformation in the DFD a new object(class) element is added to the OD.

```
16 dfdModel.ownedElement.select(lambda ts:
17    (ts.oclIsKindOf(DataTransformation) or
18    ts.oclIsKindOf(DataStore)) and
19    classDiag.addClass(name=ts.name))
```

We mention that although OCL is specified as a declarative language, the Python `lambda` functions allow us to use OCL-like constructs in an imperative manner (lines 19, 38, 48, 49).


IEEE COMPUTER SOCIETY

Once the a number of classes are added to the OD, the script draws the associations among them. First we analyze those data flows that have as source and target a data transformation and select them from the model (lines 30-39). For each pair of source-target data transformations, by using the function `addAssoc()` we identify their corresponding classes in the OD model (lines 20-26) and add an association (lines 20-28). Upon adding a new association, the target class receives a method and preserves the initial name of the data flow.

```
20 def addAssoc(source,destination,theName):
21    umlModel.ownedElement.select(lambda src:
22       src.oclIsKindOf(Class) and
23       src.name==source.name and
24       umlModel.ownedElement.select(lambda dest:
25          dest.oclIsKindOf(Class) and
26          dest.name==destination.name and
27          classDiag.addAssocociation(
28             src, dest, name=theName+"()")))
29    return 1
30 dfdModel.ownedElement.select(lambda f:
31    f.oclIsKindOf(DataFlow) and
32    dfdModel.ownedElement.select(lambda src:
33       src.oclIsKindOf(DataTransformation) and
34       f.connection[0] in src.association and
35       dfdModel.ownedElement.select(lambda dst:
36          dst.oclIsKindOf(DataTransformation) and
37          f.connection[1] in dst.association and
38          addAssoc(src,dst,
39            "send"+string.split(f.name,'+')[0]))))
```

Classes originating from data stores receive as attributes the parameter of the data flows and corresponding methods to read and/or write those parameters. Below we only present the addition of a `read()` method for a given class.

```
40 dfdModel.ownedElement.select(lambda f:
41    f.oclIsKindOf(DataFlow) and
42    topDfd.ownedElement.select(lambda src:
43    src.oclIsKindOf(DataStore)and
44    f.connection[0] in src.association and
45    topDfd.ownedElement.select(lambda dst:
46      dst.oclIsKindOf(DataTransformation) and
47      f.connection[1] in dst.association and
48      addAssoc(src,dst,"read"+string.split(f.name,'+')[0])
49      and addAttr(src,string.split((f.name,'+')[0]))))
```

The following function adds an attribute with the name specified by `att` to the class corresponding to the `flow` data flow.

```
50 def addAttr(flow, attN):
51    className=string.split(flow.name, '+')[0]
52    theClass=classDiag.ownedElement.select(lambda cl:
53       cl.oclIsKindOf(Class) and cl.name==className)
54    if attN not in theClass[0].feature.name:
55       attr=Attribute(name=attN,visibility=1,type=myType)
56       theClass[0].feature.insert(attr)
57    return 1
```

A similar approach is taken to complete the remaining part of the transformation. A complete version of the transformation scripts can be found in [26].

## 6. Conclusions

In this paper we have shown how, by using a model-based approach, we integrated the data-flow and object-oriented paradigms for the specification and design of embedded systems. Both views are useful due to the different perspectives they provide on the system under consideration. We have also shown that, by using models, it is possible to create and manipulate artifacts provided by both DFD and object oriented paradigms, and also to implement automated transformations between the different steps of the design process.

We must state though, that the automated transformations have to be seen more as an aid to the designer and not something to replace him or her. Since both models (UML and DFD) have a user-friendly view by using the SMW tool, the presented approach supports the human (i.e. designer) intervention. We mention that the transformations presented in the paper are not complete examples due to space reasons. A more complete version and more implementation details can be found in [26].

The way the diagrams were created and transformed allows the designer to trace what pieces of functionality different elements in the system belong to. Although not evident in Fig. 9 because of typographical reasons, the elements in Fig. 2, 3 and 4 bear tags that are preserved from one transformation to another. For instance, in Fig. 4 we can trace objects {2.c} and {2.i} as belonging to the Send Error {2.} use case in Fig. 2.

Moreover, we mention that 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. For instance, the object diagram in Fig. 4 can be easily transformed into a DFD similar to the one in Fig. 2 even if the designer adds new internal details for some of the objects. To be able to perform the reverse transformation, one has to avoid object refactoring.

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. The same combination appears to occur, at an orthogonal perspective, with specification and implementation [25]. Our approach promotes a mapping between structural (object diagram) and dynamic (DFD) models. If a proper balance between the 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, being thus one of them useless.

For the moment, since we were addressing protocol processing applications targeted to hardware platform implementations we have intentionally avoided referring to specific object-oriented mechanisms as inheritance and polymorphism. Their dynamic nature is in contrast with the statical nature of the hardware components. Future work includes investigating a more rigorous method for data classi-


IEEE COMPUTER SOCIETY

fication inside DFDs, that will maximize benefits of object-oriented mechanisms like inheritance and polymorphism. For instance, we can see that the routingDatagram class in Fig. 9 looks similar with a parent class of the responseDatagram and requestDatagram.

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 [1].

## 7. Acknowledgements

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

## References

[1] http://www.abo.fi/~dtruscan/ipv6index.html.

[2] *Open SystemC Initiative*. http://www.systemc.org.

[3] B. Alabiso. Transformation of Data Flow Analysis Models to Object Oriented Design. In *OOPSLA '88*, pages 335–53. ACM Press, 1988.

[4] M. Alanen, J. Lilius, I. Porres, and D. Truscan. Realizing a Model Driven Engineering Process. Technical Report 565, Turku Centre for Computer Science (TUCS), Turku, Finland, Nov. 2003.

[5] L. B. Becker, M. Gergeleit, E. Nett, and C. E. Pereira. An Integrated Environment for the Complete Development Cycle of an Object-Oriented Distributed Real-Time System. In *2nd IEEE Intl Symp. on Object-Oriented Real-Time Distributed Computing*, pages 165–71. IEEE CS Press, May 1999.

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

[7] H. Corporaal. *Microprocessor Architectures - from VLIW to TTA*. J. Wiley and Sons Ltd., West Sussex, England, 1998.

[8] O. Dieste, M. Genero, N. Juristo, J. Maté, and A. Moreno. A Conceptual Model Completely Independent of the Implementation Paradigm. *The Journal of Systems and Software*, 68(3):183–198, 2003.

[9] D. Dori. *Object-Process Methodology — A Holistic Systems Paradigm*. Springer Verlag, 2002.

[10] J. M. Fernandes and J. Lilius. Functional and Object-Oriented Modeling of Embedded Software. In *11th Intl. Conf. and Workshop on the Engineering of Computer Based Systems (ECBS'04)*, May 2004.

[11] J. M. Fernandes, R. J. Machado, and H. D. Santos. Modelling Industrial Embedded Systems with UML. In *Proceedings of CODES 2000*, pages 18–22. ACM Press, San Diego, CA USA, May 2000.

[12] H. Gall and R. Klösch. Finding Objects in Procedural Programs: An Alternative Approach. In *2nd Working Conference on Reverse Engineering*, pages 208–16. IEEE CS Press, July 1995.

[13] J. Isaksson, D. Truscan, and J. Lilius. A MOF-based Metamodel for SA/RT. Technical Report 555, Turku Centre for Computer Science (TUCS), Turku, Finland, Oct. 2003.

[14] I. Jacobson and F. Lindström. Reengineering of Old Systems to an Object-Oriented Architecture. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 340–50. ACM Press, 1991.

[15] Kiczales et al. Aspect-Oriented Programming. In *ECOOP '97 - Object-Oriented Programming*, volume 1241 of *LNCS*, pages 140–9. Springer-Verlag, 1997.

[16] T. Korson and J. D. McGregor. Understanding Object-Oriented: A Unifying Paradigm. *Communications of the ACM*, 33(9):40–60, Sep. 1990.

[17] J. Lilius and D. Truscan. UML-driven TTA-based Protocol Processor Design. In *Forum on specification and Design Languages (FDL '02)*, Sep. 2002.

[18] OMG. OMG Model Driven Architecture, July 2001. Document ormsc/2001-07-01, http://www.omg.org.

[19] M. Peleg and D. Dori. Extending the Object-Process Methodology to Handle Real-Time Systems. *Journal of Object Oriented Programming*, 11(8):53–8, Jan. 1999.

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

[21] I. Porres. A Toolkit for Manipulating UML Models. *Software and Systems Modeling, Springer-Verlag*, 2(4):262–277, Dec. 2003.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

[23] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Developement. *IEEE Software*, 20(5):42–45, Sep/Oct 2003.

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

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

[26] D. Truscan, J. M. Fernandes, and J. Lilius. Tool support for DFD to UML model-based transformations. Technical Report 519, Turku Centre for Computer Science (TUCS), Turku, Finland, Apr. 2003.



IEEE COMPUTER SOCIETY