Reconceptualization of Class-based Representation in UML

Sabah Al-Fedaghi

Computer Engineering Department, Kuwait University
P.O. Box 5969, 13060, Kuwait

Abstract

The requirement phase in the software development process is typically formulated using UML diagrams, including use cases and conceptual class diagrams. It is claimed that UML is suitable for modeling at the domain level; accordingly, many enhancements to these diagrams have been proposed to achieve a more comprehensive representation of functionality of the system from the conceptual (computation-independent) point of view. This paper proposes a uniform conceptual methodology that integrates static and dynamic features to provide a foundation for system design in the next phase of development. UML-based modeling and this new methodology are contrasted in examples that demonstrate the feasibility of the new approach for use in formulating system requirements.

Keywords: *Software development, requirement phase, conceptual model, UML, conceptual class diagram.*

1. Introduction

An information system serves a real application and reflects the reality of the static structure and dynamic activities of organizations. Consequently, the process of developing an information system begins by drawing a domain model of the enterprise as part of the real world. The result is a conceptual description that does not include computation-dependent aspects. It serves as a means of communication and a guide for the subsequent design phase.

Modeling is a fundamental instrument used in developing a software system. In this context, many issues arise concerning quality, accuracy, completeness, and consistency of the model used. The Unified Modeling Language (UML) [1, 2] is a visual modeling language that is used to specify, construct, and document systems. Researchers have examined and proposed extending the use of object-oriented languages such as UML at the conceptual level (e.g., [3, 4, 5]).

UML has been used for conceptual/domain modeling, which is concerned with providing a representation of "things" that exist and activities that emerge in a business environment. According to current thinking, "UML is suitable for conceptual modeling but the modeler must take special care not to confuse software aspects with aspects of the real world being modelled" [6]. The problem with extending object-oriented models and languages is "that such languages possess no real-world business or organizational meaning; i.e., it is unclear what the constructs of such languages mean in terms of the business" [6]. The object-oriented design field deals with objects and attributes, while the real-world domain is formed from things and their interactions [7].

In UML, relationships identify the semantic ties between model elements and include associations, dependencies, generalizations, realizations, and transitions. With the development of UML 2.0, "several new concepts and notations have been introduced, e.g., exceptions, collection values, streams, loops, and so on" [8]. UML 2.2 offers 14 types of diagrams, including activity, class, sequence, and communication diagrams. These diagrams represent multiple system viewpoints. In addition to their value in documentation, the diagrams can be very effective for communicating and facilitating understanding, and for establishing a vision early in the design phase.

Specifically, requirements are formulated using use cases and conceptual class diagrams. The class diagram is the most fundamental and widely used UML diagram [9]. It describes a static structure that includes objects and relationships between them. It is also used for both conceptual/domain modeling and detailed design modeling, formulated using use cases and conceptual class diagrams.

The class diagram is "a central modeling technique that runs through nearly all object-oriented methods" [10]. According to Ambler [11],

> UML 2 class diagrams are the mainstay of object-oriented analysis and design. UML 2 class diagrams show the classes of the system, their interrelationships…, and the operations and attributes of the classes…
> Class diagrams are typically used … to: explore domain concepts in the form of a domain model, analyze

requirements in the form of a *conceptual*/analysis model, and depict the detailed design of object-oriented or object-based software. [11] (Italics added)

Such a claim reflects the main paradigm in development of information systems where class diagrams (or similar diagrams, e.g., entity-relationship diagrams) form the static structure upon which all other aspects (e.g., dynamic, constraints) are built. This paradigm may have originated in the classic way of thinking about building a physical system, starting with a static description that identifies basic components and subcomponents and their interrelationships and attributes. Though Amber [11] mentions "operations" in the quote above, it seems that this term does not mean the actual dynamic behavior of the system. Behavioral aspects are emphasized in another type of description such as activity diagrams.

The conceptual class diagram and its associated use cases are "not rich enough for generating the prototype as it does not provide information about the *flow* of interacting events between the actors and the system when carrying out a use case" [12] (Italics added). Conceptual class diagrams represent concepts that "naturally relate to the classes" with no regard for software implementation [10, 13]. It is not clear whether this means equating *concepts* with *classes*. According to [10], "Unfortunately the lines between the [conceptual, specification, and implementation] perspectives [when using use class diagrams] are not sharp, and most modelers do not take care to get their perspective sorted out when they are drawing." Generally, "the biggest danger with class diagrams is that you can get bogged down in implementation details far too early. To combat this, use the conceptual [perspective]" [10].

Many proposals have been made for development of a methodology to fill the need for more comprehensive representation of functionality of the system from the conceptual (computation-independent) point of view. For example, according to [14],

> Use cases are a notation not an approach. Their usage is not systematic in comparison with systematic approaches that enable identification of all system requirements. Creation of use case models and establishment of concepts and relations among them are usually rather informal than semiformal… Use cases' fragmentary nature does not give any answer to questions about identifying all of system's use cases, conflicts among the use cases, gaps in the system's requirements, how changes can affect behavior that other use cases describe … Use cases must be applied as *a part of* a technique, whose first activity is a construction of a well-defined problem domain model. [15]

Osis et. al. [14] use "a goal-based method" to define a use case model and graph transformation from "topological functioning modeling" to a conceptual class to enable "the definition between domain concepts and their relations to be established." A conceptual class model is detailed to the level where it uses only one type of object.

Another approach is to incorporate Object Role Modeling, or ORM:

> Despite its upcoming inception as the world standard for expressing the results of the conceptualization and specification of the aforementioned types of systems, UML is considered to be : 'incomplete, inconsistent and unnecessarily complex' [16].

According to Bollen [17],

> This incompleteness, inconsistency and complexity, however, can be avoided when a conceptual schema design procedure from a fact-oriented modeling approach will be applied on data use cases [16]. The resulting conceptual schema will provide a 'semantic-rich' starting point for the creation of a UML class diagram. [17]

We claim that the conceptual class diagram in UML lacks a fundamental notion, thus causing conceptual fragmentation. *Classes* (objects) need a bonding mechanism (e.g., *flow* among them) to capture static and dynamic continuity in the total conceptual picture.

This paper proposes a new approach to the problem of conceptual representation of functionality in the field of object oriented software development. Instead of the class/object-based description of requirements, the methodology incorporates the dynamic aspects of the system by adopting the notion of *flow*.

2. Motivational Example

Ambler [11] discusses the notion of "conceptual class diagrams" in the example shown in Fig. 1 depicting the conceptual model of a university.

> I could have added an attribute in the Seminar class called Waiting List but, instead, chose to model it as an association because that is what it actually represents: that seminar objects maintain a waiting list

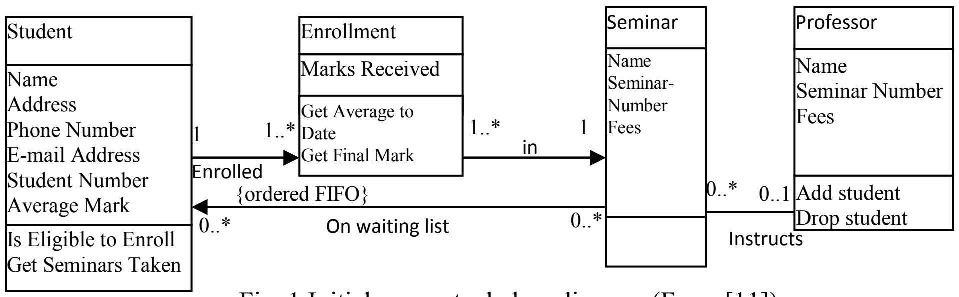

Fig. 1 Initial conceptual class diagram (From [11])

> of zero or more student objects. Attributes and associations are both properties in the UML 2.0 so they're treated as basically the same sort of thing… I prefer to keep my models simple and assume that the attributes and operations exist to implement the associations. [11]

The operation called *enrolled* would be engaged to calculate a student's average mark and provide information about seminars taken.

There is also an *enrolled* in the association between Enrollment and Seminar to produce a list of seminars taken. {ordered FIFO} is a constraint on the association between Seminar and Student.

> Following a consistent and sensible naming convention helps to make your diagrams readable. Notice my use of question marks in the note. My style is to mark unknown information on my diagrams this way to remind myself that I need to look into it. [11]

While this brief description is not a complete account of the example, it is sufficient for our purpose of showing the general flavor of this methodology. It reflects a static structure with strict exclusion of dynamic aspects that are modeled by other diagrams.

Our strategy is to contrast this depiction of the involved application of our flow-based conceptual picture on the basis of a systematic method that integrates static and dynamic features. To achieve this contrast, and to make the paper self-contained, the next section briefly reviews the basic concepts of the model, called the Flowthing Model, used in this method as introduced in several papers [18, 19, 20, 21].

3. Flowthing Model

The Flowthing Model (FM) is a uniform method for representing things that flow, called flowthings. Flow in FM refers to the exclusive (i.e., being in one and only one) transformation among six states (also called stages) of transfer, process, create, release, arrive, and accept.

To exemplify FM, consider flows of a utility such as electricity in a city. In the power station, electricity is *created* then *transferred* to city substations through transmission lines, where it *arrives*. The substations are safety zones where electricity is *accepted* if it is of the right type (e.g., voltage); otherwise it is cut off. Electricity is then *processed*, as in the case of *creating* different voltage values to be sent through different feeders in the power distribution system. After that, electricity is *released* from the distribution substation to be *transferred* to homes. For the flowthing, in this case electricity, FM asserts that only six mutually exclusive states exist: transferred, arrived, accepted, processed, created, and released, as shown in Fig. 2. This diagram is called a flowsystem.

All other states of flowthings are not generic states. For example, we may have *stored created* flowthings, *stored processed* flowthings, *stored received* flowthings, etc. Flowthings can be released but not transferred (e.g., the channel is down), or arrived but not accepted, … We use *Receive* as a combined stage of *Arrive* and *Accept* whenever appropriate, i.e., whenever arriving flowthings are always accepted. The fundamental elements of FM are described as follows:

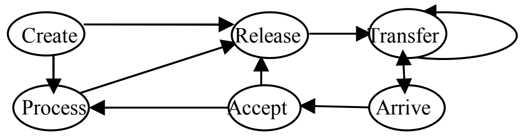

Fig. 2 Flowsystem

Flowthing: A thing (e.g., information, material, money, shuttle, good) that has the capability of being created, released, transferred, arrived, accepted, and processed while flowing within and between systems.
A flow system (referred to as flowsystem), as depicted in Fig. 2, comprises the internal flows (solid arrows) of a system with the six stages and transactions among them.
Spheres and subspheres are the environments of the flowthing, such as
- Computer electronics with signals as flowthings
- Human mind with information as flowthings
- Organization information system with records as flowthings

A sphere (or subsphere) can incorporate many flowthings into its flowsystems, e.g., a computer includes an information (abstract objects) flowsystem and an electronic signal (physical objects) flowsystem.
Triggering is a transformation (denoted by a dashed arrow) from one flow to another, e.g., flow of electricity triggers the flow of air.
Notice that we use a single rectangle when a sphere comprises a single flowsystem.

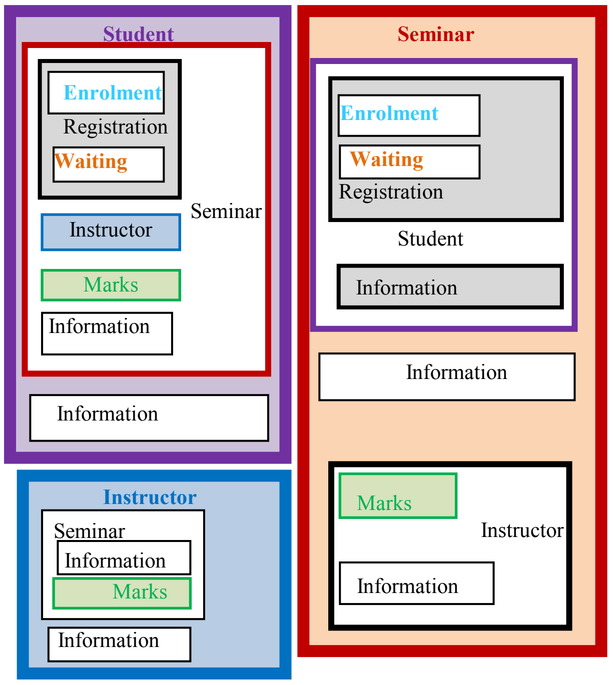

Fig. 3 The spheres in the given example

3. Redrawing the Motivational Example

The conceptual landscape of Ambler's [11] example presents the hierarchy of the spheres, as shown in Fig. 3. There are three main spheres: Student, Seminar, and Instructor. The Student sphere has a Seminar subsphere and an Information subsphere, and the Seminar sphere has a Student subsphere and an Information subsphere.

The *Seminar* in *Student* (for simplicity's sake, we do not always repeat the "sphere" and "subsphere" qualifiers) represents the *Seminar* from the point of view of the *Student*. Let us denote this as *Student.Sphere*. Student.Sphere includes all parts of the sphere known by Student. In reality, student concern about a seminar covers enrolment, waiting, instructor, marks, and information (e.g., title, prerequisites) about the seminar. We have included Registration in the figure as a super-sphere of Enrolment and Waiting because this seems to encompass the functionality of these two concepts. Such an action is analogous to a person conceptualizing a "sleeping place" with "walk-in closet," immediately bringing to mind the word "bedroom" as an encompassing term.

Notice that we follow the original view of the example, associating *Student* directly with *Seminar* and indirectly with *Instructo*r. Figure 4 shows a general picture of flows among spheres and subspheres.

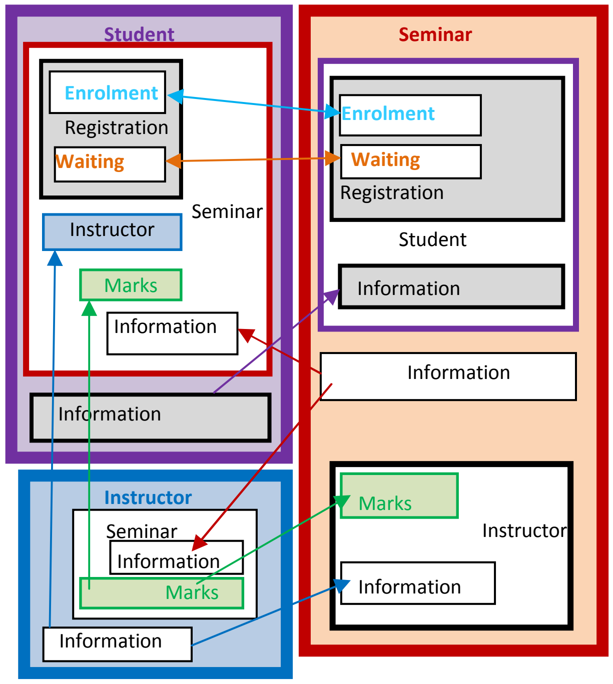

Fig. 4 Flows among spheres and subspheres

Fig. 5 depicts the conceptual representation of the example. Starting with circle 1 in Fig. 5, in the Student.Enrolment flowsystem, enrolment is created that flows (2) to Seminar.Enrolment, where it is processed (3) and stored (4). It is also possible that, in Student.Enrolement, it is required to access an already stored Enrolment. In this case, Process (5) in Student.Enrolment with input Enrolment identifier (ID) triggers (6) the retrieval of an Enrolment in Seminar.Enrolment (7) that flows (8) to Student.Enrolment, where it is processed.

To illustrate the essence of this methodology, imagine a student registering for a seminar, then going through the following possible phases:
1. At the beginning a screen appears that gives the opportunity to select Student.Regisration.Enrolment
2. When Student.Enrolment is selected, the screen in Fig. 6(a) appears, giving a choice between new enrolment (1 in Fig. 5) or Processing an already existing enrolment (5 in Fig. 5).

3. Suppose that New is selected; the screen shown in Fig. 6(b) then appears, with the options to create, release, or transfer a new enrolment.

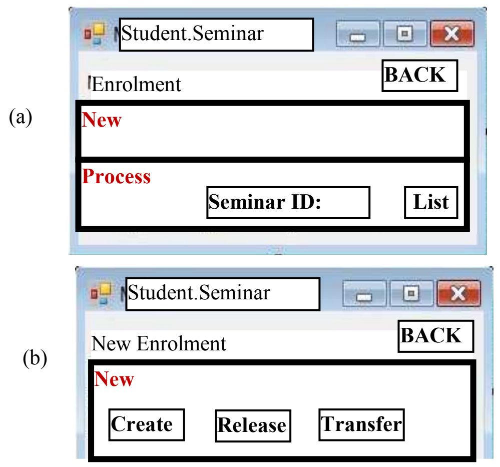

Fig. 6 Possible Screen operation in Student sphere

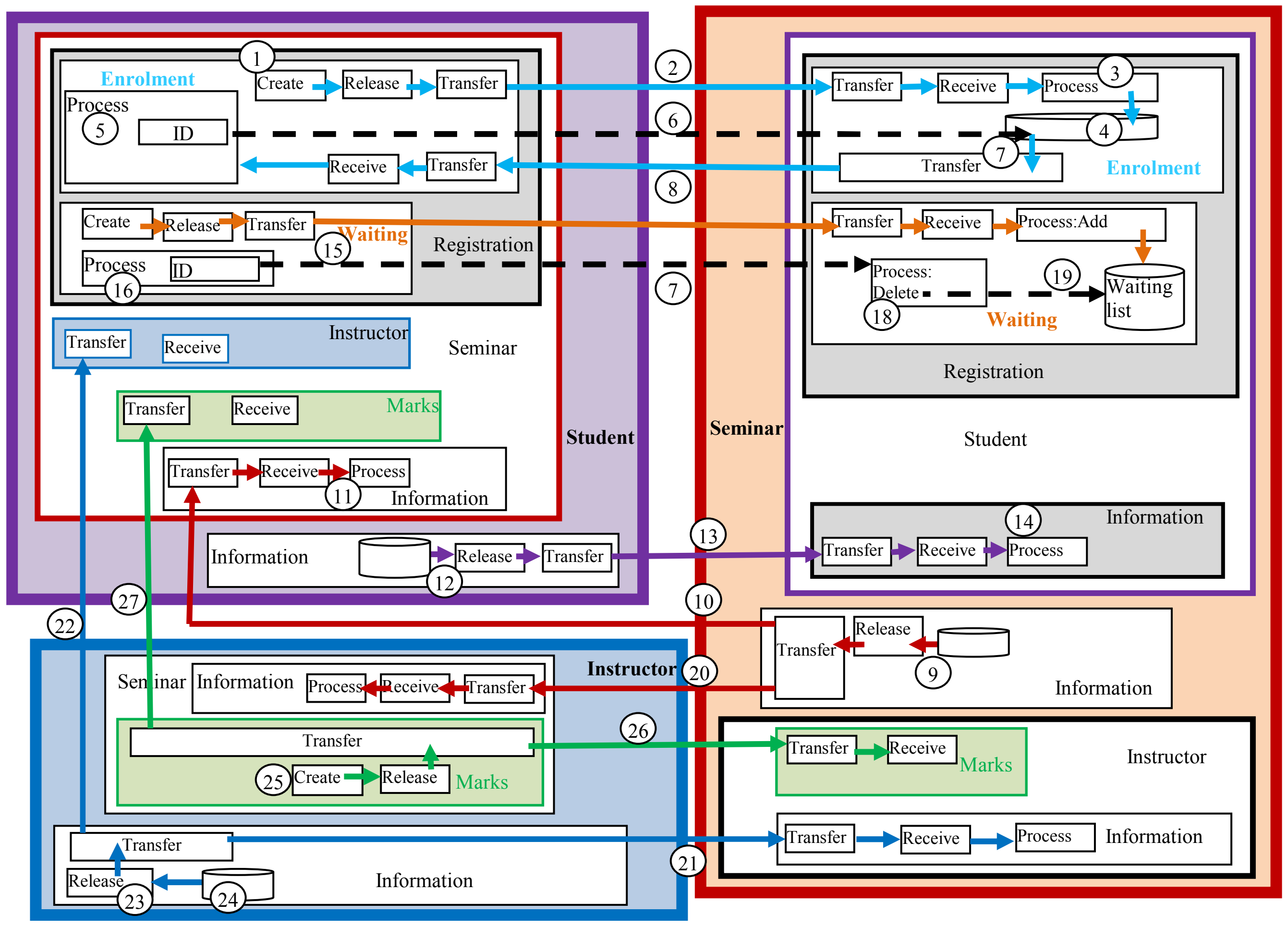

Fig. 5 FM conceptual representation of the example

4. Suppose that Create is selected; the screen shown in Fig. 7(a) then appears. A rolling window gives the capability of selecting a seminar. Note that because of flow of information from/to Student.Seminar, this set of screens is additionally mapped to the two flows shown in Fig. 5:
- Retrieval of seminar information (9) that flows (10) to be processed in Student.Seminar (11), and
- Retrieval of student information (12) that flows (13) to be processed in Student.Seminar (14).
5. If Process in Fig. 6(a) is selected, the user then provides the required seminar ID using List in the figure, which gives a list of all available seminars. The screen in Fig. 7(b) appears, where the name of the selected seminar appears and a previous registration can be dropped.

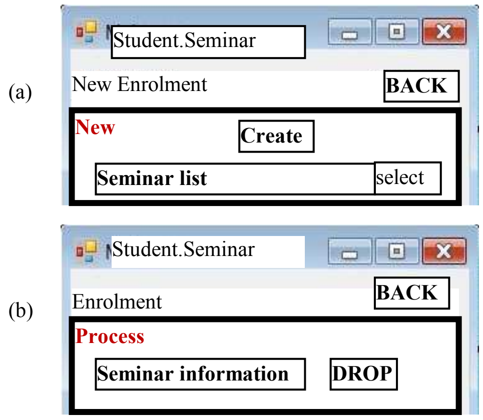

Fig. 7 Another possible Screen in Student sphere

Note the systematic mapping screens (Figs. 6 and 7) reflecting the underlying flow (Fig. 5) and utilizing the same basic FM flowsystems.

Returning to Fig. 5, Waiting (15) is modeled in a similar manner to Enrolment; however, its processing (16) triggers (17) the process of Delete (18), which in turn triggers (19) deletion of an entry in the waiting list.

The Instructor sphere also receives information about the Seminar (20), and information about Instructor flows to Seminar (21) and Student (22). Note the information flows that can be specified *inside Release* (23). For example, Information about Instructor that flows to Student includes only the name, while the information that flows to Seminar may include other data. Also, in Fig. 5, it is assumed that the Information about Instructor is already stored (23). It is possible to add Create in Instructor.Information to build a functionality (e.g., screen) that would enable a user of Instructor to input data about instructors.

The FM representation maps flows of different flowthings, analogous to a map of flows of rivers, streams, and channels in an irrigation system. The model portrays basic infrastructure over which constraints, rules, redirections (e.g., logical operations: AND, OR, …; security; synchronization, timing, and so forth) can be superimposed.

Marks are created in Instructor.Marks (24) and flow to Seminar (26) and Student (27). It is possible to redraw the flow of Marks such that it does not flow directly to Student but instead reaches Student through Seminar after processing (e.g., checked first by Seminar user).

Contrasting the conceptual class diagram (Fig. 1) with the FM representation (Fig. 5), it is clear that the FM diagram is a purely conceptual depiction that is not infected with data record–based thinking. Thus, it is a neutral picture that can be used for general understanding among technicians, administration, and users.

The FM depiction can easily be enriched with additional agreed-on information regarding constraints, security, … as an integrated part of the description (e.g., details within process, create, … boxes), or as annotations over the map since it a more comprehensive representation of functionality of the system from the conceptual (computation-independent) viewpoint.

The UML class diagram lacks a fundamental connecting notion to act as a conceptualization instrument tying the description together. In FM, flows tie spheres; thus, static and dynamic aspects build the system. Class diagrams reflect conceptual blurriness because of a static base; thus, they obstruct the ability to differentiate types of flows (an arrow can represent many things). The FM flow-based description differentiates diverse types of flows.

3. Dynamic Modeling

Sequence diagrams, besides other diagrams, are used in UML for capturing dynamic aspects. "Sequence diagrams, along with class diagrams and physical data models are … the most important design-level models for modern business application development" [22].

Scott [22] gives Fig. 8 (shown partially), depicting a UML sequence diagram for Enroll in the University use case. It models the detailed logic at the object-level. Messages are indicated as labeled arrows; when the source and target of a message are an object or class, the label is the method invoked. If either the source or target is a human actor, the message is labeled with brief text. Return values are optionally indicated using a dashed arrow. Stereotypes throughout the diagram, e.g., <<UI>>, represent an actor, a controller class, or a user interface (UI).

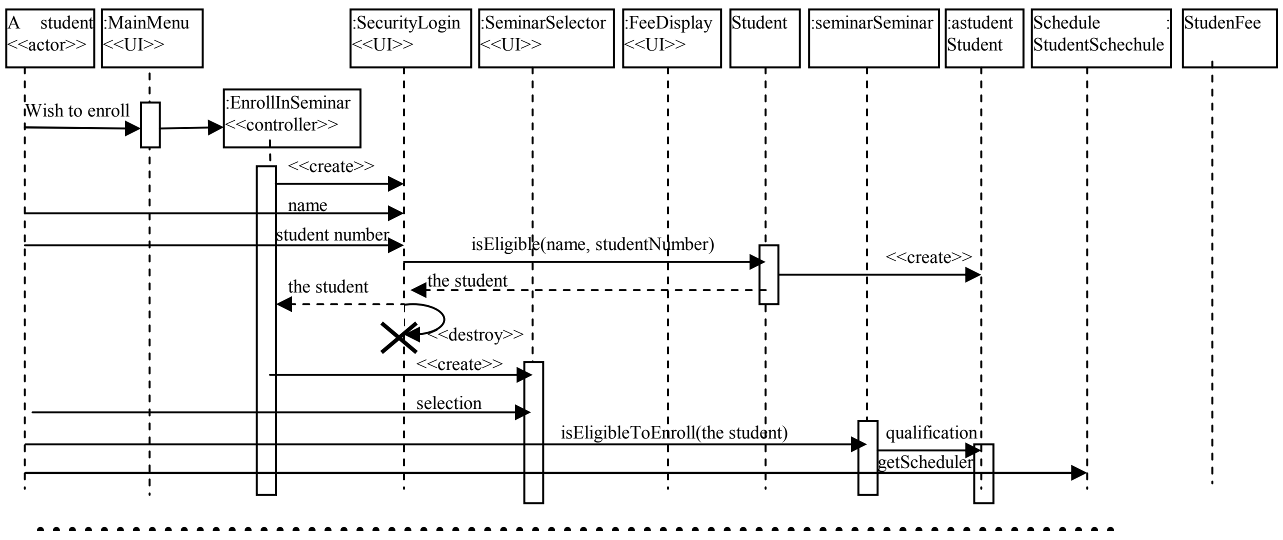

Fig. 8 Basic course of action for Enroll in Seminar use case (partial, from [22])

The creation and destruction of messages are denoted with the stereotypes <<create>> and <<destroy>>, respectively.

Again, this brief description is not a comprehensive account of the example, but it is sufficient for our purpose of showing the general flavor of the methodology of modeling dynamic behaviors of the system. We can observe the abrupt change in diagrammatization style from the class diagram to the sequence diagram, which also involves classes and objects (top of Fig. 8). It seems that the difficulty in following the sequence of events might have motivated Scott [22] to write the events in semi-pseudo language, as follows:

1. Student indicates wish to enroll
2. Student inputs name and number
3. System verifies student
4. System displays seminar list
5. Student picks seminar
6. System determines eligibility to enroll
7. System determines schedule fit
8. System calculates fees
9. System displays fees
10. System verifies student wishes to enroll
11. Student indicates yes
12. System enrolls student in seminar [22]

In contrast, the FM methodology provides a uniform treatment in the modeling of a system's dynamics. Applying the same flow in a flowsystem as in Fig. 5, for the process of enrolling in a seminar, Fig. 9 shows the resultant FM representation.

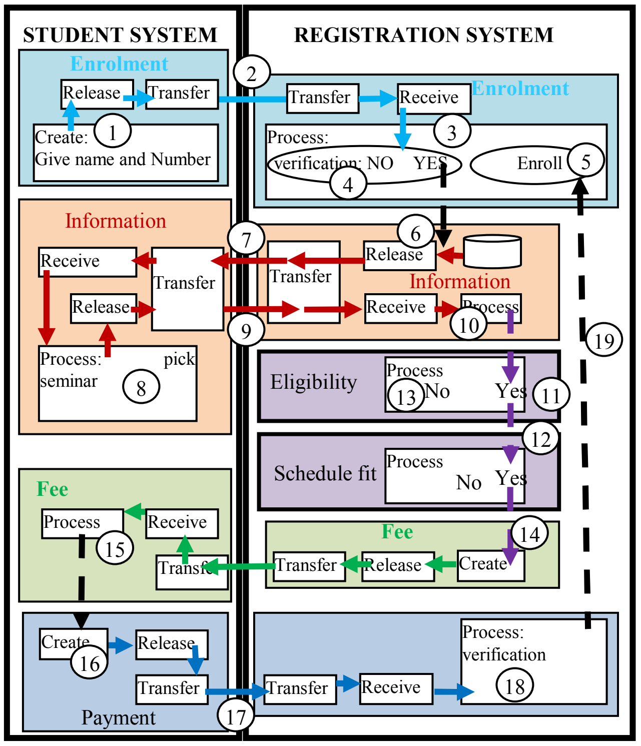

Fig. 9 The FM representation of the flow for Enroll in a seminar

In Fig. 9, a student applies for enrolment giving his/her name and Number (circle 1). These data flow (2) to the system where they are received and processed (3). Processing in the Enrolment flowsystem includes two kinds of processes: verification and enrolment. Data coming from Student are verified (4) and if data are okay (Yes in the figure), it triggers (6) the retrieval of information about Seminars that is then sent (7) to Student. Here, it not clear from the original description what to do when data are not okay (NO in process verification). The modeler can make a response of NO trigger some type of procedure such as sending of a message to Student.

When Student receives Seminar information and picks a seminar (8), this selection of a seminar flows (9) to System to be processed (10) and triggers a check for eligibility. Here again, it is not clear what this eligibility check entails. It might depend on information about the student (e.g., GPA); in this case, such information may be required to complete the eligibility check. Or, eligibility may require more information related to the seminar (e.g., prerequisite), or both types of data. Consequently, we have left the Eligibility details (and the next phase of Schedule fit) without details. Nevertheless, Process in Eligibility results in YES or NO. If YES (11), then this triggers (12) Process in Schedule fit. Again it is not clear from the original description what to do if NO occurs in Eligibility (13) or in Schedule fit. We assume that everything is okay; hence Fee data is calculated (14) and sent to Student. Upon the processing (15) of Fee data, we assume that the student makes (16) payment that flows (17) to the system. This point is also not clear from the original description, and has been added as a step. When payment is processed and verified (18), this triggers actual enrolment of the student (19).

Again it is sufficient to contrast the resulting representation of the UML methodology and the FM-based depiction of the same problems.

4. Conclusions

This paper has examined the problem of the need for development of a comprehensive representation of functionality of the system from the conceptual (computation-independent) point of view. A new methodology is proposed that uniformly integrates static and dynamic features. It is contrasted with UML class-based diagramming through examples that demonstrate the feasibility of the new approach for formulating system requirements.

We can observe that huge development efforts have been invested in the UML methodology; nevertheless, this investment should not discourage new research into alternative methods. In any event, the FM modeling technique is still in need of a great deal of work to reach a mature level as a tool for use in developing software.

Further research will apply the FM methodology for other areas such as various ULM diagrams [23], requirement specification applications [24], and software Visualization [25].

References

[1] UML 2.0 Superstructure Specification (formal/05-07-04). Object Management Group, 2005.
[2] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual, Second Edition. Addison-Wesley, 2004.
3] J. Brüning, M. Gogolla, and P. Forbrig, "Modeling and Formally Checking Workflow Properties Using UML and OCL," in P. Forbrig and H. Günther (eds.), Proc. 9th Int. Conf. Perspectives in Business Informatics Research, pp. 130–145, Berlin: Springer, LNBIP 64, 2010.
[4] I. Castillo, F. Losavio, A. Matteo, and J. Boegh, "Requirements, Aspects and Software Quality: the REASQ model", Journal of Object Technology, Vol. 9, No. 4, 2010, pp. 69–91. http://www.jot.fm/issues/issue_2010_07/article4.pdf
[5] J. Evermann, and Y. Wand, "Towards ontologically based semantics for UML constructs", In H. Kunii, S. Jajodia, and A. Solvberg, A. (eds.), Proceedings of the 20th international conference on conceptual modeling, Yokohama, Japan 2001. http://www.mcs.vuw.ac.nz/~jevermann/EvermannWandER01.pdf.
[6] J. Evermann, "Thinking ontologically: Conceptual versus design models in UML", in M. Rosemann, P. and Green (eds.), Ontologies and Business Analysis, Idea Group Publishing, 2005. http://www.mcs.vuw.ac.nz/~jevermann/EvermannChapter05.pdf
[7] E. Coatanéa, "Conceptual modelling of life cycle design: A modelling and evaluation method based on analogies and dimensionless numbers", Doctoral dissertation, Helsinki University of Technology, 2005. http://lib.tkk.fi/Diss/2005/isbn9512278537/isbn9512278537.pdf
[8] H. Storrle, and J. Hausmann, "Towards a formal semantics of UML 2.0 activities," German Software Engineering Conference, 2005. http://wwwcs.uni-paderborn.de/cs/ag-engels/Papers/2005/SE2005-Stoerrle-Hausmann-ActivityDiagrams.pdf
[9] M. Szlenk, "Formal Semantics and Reasoning about UML Class Diagram", in Proceedings of the International Conference on Dependability of Computer Systems, pp. 51–59, IEEE Computer Society, 2004. http://staff.elka.pw.edu.pl/~mszlenk/pdf/Formal-Semantics-Reasoning-UML-Class-Diagram.pdf

[10] D. Stotts, "Documenting an OO Design: Class Diagrams", 2007. http://www.cs.unc.edu/~stotts/145/CRC/class.html

[11] S. W. Ambler, "UML 2 Class Diagrams", Ambysoft Inc., 2003-2010. http://www.agilemodeling.com/artifacts/classDiagram.htm#ConceptualClassDiagrams

[12] Y. Wei, X. Li, Z. Liu, and J. He. Automatic Transformation from Requirements Models to Executable Prototypes. UNU-IIST Report No. 329. http://iist.unu.edu/www/docs/techreports/reports/report329.pdf

[13] S. Cook, and J. Daniels, Designing Object Systems: Object-oriented Modeling with Syntropy, Hemel Hempstead, UK: Prentice Hall International, 1994.

[14] J. Osis, E. Asnina, and A. Grave, "Formal Computation Independent Model of the Problem Domain within the MDA", ISIM 2007. http://ceur-ws.org/Vol-252/paper06.pdf

[15] S. Ferg, "What's Wrong with Use Cases?" 2003. http://www.jacksonworkbench.co.uk/stevefergspages/papers/ferg--whats_wrong_with_use_cases.html

[16] T. Halpin, "Augmenting UML with Fact-orientation", in Workshop Proceedings: UML: A Critical Evaluation and Suggested Future, HICCS-34 Conference, 2001.

[17] P. Bollen, "A Formal ORM-to-UML Mapping Algorithm", Research. Memoranda 015, University of Maastricht, The Netherlands, 2002. http://arno.unimaas.nl/show.cgi?fid=465

[18] S. Al-Fedaghi, "Scrutinizing UML Activity Diagrams", 17th International Conference on Information Systems Development (ISD2008), Paphos, Cyprus, August 25-27, 2008.

[19] S. Al-Fedaghi, "Interpretation of Information Processing Regulations", Journal of Software Engineering & Applications, Vol. 2, No. 2, pp. 67-76, 2009.

[20] S. Al-Fedaghi, "A Conceptual Foundation for the Shannon-Weaver Model of Communication", International Journal of Soft Computing, Vol. 7, No. 1, 2012, pp. 12-19.

[21] S. Al-Fedaghi, "Awareness of Context and Privacy", American Society for Information Science & Technology (ASIS&T) Bulletin, Vol. 38, No. 2, 2011.

[22] S. W. Ambler, UML 2 Sequence Diagrams, 2003–2010. http://www.agilemodeling.com/artifacts/sequenceDiagram.htm

[23] R. Elmansouri, H. Hamrouche and A. Chaoui, "From UML Activity Diagrams to CSP Expressions: A Graph Transformation Approach using Atom Tool, International Journal of Computer Science Issues, Vol. 8, Issue 2, March 2011.

[24] L. Raamesh and G. V. Uma, "Reliable Mining of Automatically Generated Test Cases from Software Requirements Specification (SRS)", International Journal of Computer Science Issues, Vol. 7, Issue 1, No. 3, January 2010.

[25] A. Anand Rao and K. Madhavi, "Framework for Visualizing Model-Driven Software Evolution and its Application", International Journal of Computer Science Issues, Vol. 7, Issue 1, No. 3, January 2010.

Sabah Al-Fedaghi holds an MS and a PhD in computer science from the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, and a BS in Engineering Science from Arizona State University, Tempe. He has published two books and more than 140 papers in journals and conferences on Software Engineering, Database Systems, Information Systems, Computer/information Ethics, Information Privacy, Information Security and Assurance, Information Warfare, Conceptual Modeling, System Modeling, Information Seeking, and Artificial Agents. He is an associate professor in the Computer Engineering Department, Kuwait University. He previously worked as a programmer at the Kuwait Oil Company and headed the Electrical and Computer Engineering Department (1991–1994) and the Computer Engineering Department (2000–2007). http://cpe.kuniv.edu/images/CVs/sabah.pdf