Extending iStar2.0 Metamodel to Define Data Structures

Xavier Franch [1]

[1] *Universitat Politècnica de Catalunya, Barcelona, Spain*

Abstract

iStar2.0 provides a recommendation for the core constructs defined in the *i** language, which are articulated around a metamodel. When applying iStar2.0 to a particular domain, it can be necessary to extend this metamodel in order to represent more specialized concepts. One of these domains is that of data structures, as implementation of abstract data types. In this paper, we build upon previous work on using *i** to describe data structures from an intentional point of view, by introducing new constructs in iStar2.0 and adding them to the iStar2.0 metamodel. We illustrate the approach using some well-known abstract data types (sequences, functions, …) and the data structures implementing them (linked lists, heaps, hash tables, …).

Keywords

iStar2.0, data structures, abstract data type, metamodel.

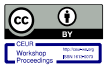
1. Introduction

In [1], we explored the use of *i** to describe abstract data types and data structures, using the iStar 2.0 language [2]. We responded to two research questions: *(i)* how can iStar 2.0 be used to describe the specification of abstract data types, and *(ii)* how can iStar 2.0 be used to describe data structures that implement those abstract data types. These questions are of interest in two contexts: *(i)* education, where teachers can use the intentional view to summarize the main high-level characteristics of abstract data types and data structures, and students may focus on this high-level view before diving into the details; *(ii)* software development, where the high-level description of abstract data types and data structures may help programmers to decide which one applies better in a particular context. In that first paper, we proposed: *(i)* the use of modules as defined in [3][4] to encapsulate specifications and implementations of abstract data types; *(ii)* the use of the iStar 2.0 *participates-in* construct to link implementations and specifications; *(iii)* the use of specialization as defined in [5] to allow building hierarchies of specifications and implementations according to their similarities and differences. These elements and their application to the two research questions were described informally. In this paper, we advance the results of [1] in two directions:

1. We elaborate some of the decision taken in [1] about the form that the proposed new constructs need to take.
2. We extend the iStar 2.0 metamodel defined in [2] with all these constructs and their integrity constraints.

2. Constructs to be added to the iStar 2.0 metamodel

Although at a first glance it may seem counter-intuitive to use *i** to describe data structures, we argue that we may find similar situations in other *i**-related research lines. One of the most investigated areas in this context is the use of *i** to model and reason about software architectures: software components are modelled as actors and their connections are modelled as dependencies (e.g., a goal to be fulfilled, a message to be sent as part of a call, a method to be offered to the outside) [6]. Similar

Proceedings of the 14th International iStar Workshop, October 18-21, 2021, St. Johns (NL), Canada
EMAIL: franch@essi.upc.edu
ORCID: 0000-0001-9733-8830


approaches are applied in domains and types of systems as product lines [7], business intelligence [8] and service-oriented systems [9]. In our work on data structures, *i** is used to describe software components (the data structures and their corresponding abstract data types) with the goal of improving their understanding and ultimately, informing their selection in the context of a particular program with particular requirements, as we have done in the past also in the context of software packages [10].

Under this assumption, the constructs that we need to add to the iStar 2.0 metamodel are:

1. Modules. The need for having modules is clearly exposed in [1] and comes from the modular nature of abstract data types and data structures [11]. While in [1] we proposed the existence of only one generic type of module, we now advocate to distinguish among specification and implementation modules because: *(i)* the type of intentional elements is not the same in specifications as in implementations, *(ii)* the link among specifications and implementations is more evident given this distinction.

2. Links. In [1], we proposed to use the iStar 2.0 *participates-in* link to establish that an implementation (i.e., a data structure) implements an abstract data type. While this need prevails, we propose a new, different type of link specific for this situation, *implements*, since *participates-in* is too generic. It must be remarked that we considered the option to establish this link not at the level of actors, but at the level of modules. However, we discarded this option to provide a uniform treatment with respect to specialization (see below), which is established at the level of actors. Since one module introduces only one actor, this decision does not have a significant impact on the resulting models.

3. Specialization. The framework presented in [1] proposes specialization to structure the different abstract data types both at the level of specialization and implementation. While iStar 2.0 offers a specialization construct at the level of actors, it does not detail how this construct is refined at the level of intentional elements (as it also happened with the seminal *i** [12]). For this reason, we adopted the approach presented in [5] that fits to the needs of the data structure modeling problem.

4. Efficiency signature. In the context of data structure selection, the establishment of their asymptotic efficiency [13] traditionally plays an important role. However, this concept is quite low level (i.e., typically appearing related to code either in ad-hoc languages [14] or as annotations [15]) and it is difficult to reconcile with the intentional perspective that *i** provides. While an option could be simply getting rid of it, we think that not including it makes the proposal incomplete and forces the potential adopted of the approach to complement it with additional information. Therefore, we propose:

- A specification module should declare the parameters used for measuring the asymptotic efficiency of data structures (typically, number of elements).
- An implementation module should declare the asymptotic efficiency of those critical operations implemented following the strategy of the chosen data structure.

Figure 1 presents an example that illustrates the constructs defined above. There appear three modules, namely two specifications and one implementation. We adopt UML's package symbol to represent the modules in an intuitive form. We do not graphically distinguish between specification modules and implementation modules since we did not find any intuitive graphical way; instead, we simply use labels (even if Moody advises about the risk of doing so [16]). The two specification modules define one abstract data type each, modelled as actors, which are linked through an iStar 2.0 specialization link (*is-a*). The parent module defines the concept of *Function* as abstract data type (i.e., an abstract data type that keeps a correspondence from keys into values [17]) and three dependencies which, according to [3][4], are left open in the depender side. These dependencies establish that the depender (i.e., a software component integrating a *Function* instance) can add, remove and access to individual elements, while for that, such depender needs to provide the concept of *Key*. The module declares in addition the *Function*'s efficiency signature, which in this case only establishes the magnitude *n* representing number of elements in order to establish time efficiency later on. The child specification module declares *Mapping* as a specialization of *Function* and redefines one of the operations; according to [5], the redefinition is established by name and graphically highlighted using a color code (yellow). The semantics of the redefinition is not part of the model, but according to [18], it can be defined equationally as:

- In *Functions*, access(insert(F, <k, v>), k) = <k, v>
- In *Mappings*, look-up(insert(F, <k, v>), k) = v

We refer to [18] for more details. Therein, we may find other specializations for *Functions*, as for instance *Sets*, in which accessing is redefined as membership (also shown in Section 4).

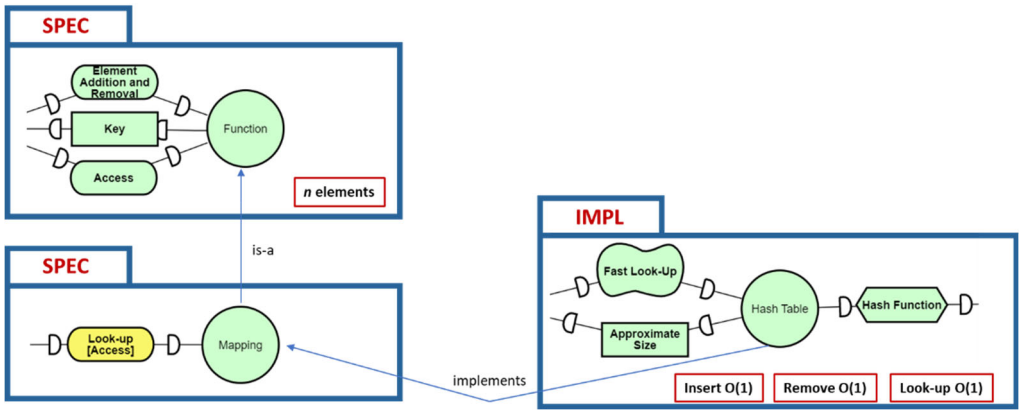


Figure 1: Example of representation of concepts needed when modeling abstract data types and data structures using extended iStar 2.0.

The implementation module shows the intentional view and efficiency signature of a particular implementation strategy for mappings, i.e. using a *Hash Table*, as established by the *implements* link among the corresponding actors. The most remarkable quality that this implementation offers is *Fast Look-Up* to the elements therein, as the efficiency signature (enclosed in a red-line rectangle) shows with a constant big-Oh asymptotic time [13]. The price to pay is that the depender needs to estimate the *Approximate Size* of the data structure and needs to provide an appropriate *Hash Function*.

3. Extending the iStar 2.0 metamodel

Figure 2 shows the extension of the iStar 2.0 metamodel with the constructs presented in Section 2, while Table 1 compiles the necessary integrity constraints. Details follow:

1. Modules. The metamodel includes an abstract class *Module* with two specializations, namely *Specification* and *Implementation*. In any case, each type of module includes only one actor and open dependencies, both incoming (what the specification or implementation offers to its customers) and outgoing (representing the requirements that the specification or implementation poses onto its customers), being both sets disjoint (**IC1**). It is worth remarking that, due to their abstract nature, specifications cannot include dependencies with qualities or tasks as dependum (**IC2**).

2. Links. The *implements* link is added to the metamodel, looking similar to the specialization link provided by iStar 2.0 (see below). Two integrity constraints (**IC3** and **IC4**) ensure that *implements* links an implementation to a specification. Multiplicities also make clear that an implementation can implement only one specification.

3. Specialization. As said, iStar 2.0 offers the *is-a* link to represent specialization at the level of actor. In addition, we add a new association linking intentional elements. We call such association *reinforces*, which is one of the three cases defined in [5] for specialization of intentional elements and is characterized by integrity constraint **IC6** (which restricts the permitted type changes of intentional elements; we refer to this article for a detailed explanation of the meaning). Besides, **IC5** forces the dependencies to which the intentional element belongs, to refer to the same actor.

4. Efficiency signature. An efficiency signature consists of a list of pairs that slightly differ depending on the type of module. In specifications, the second component of the pair is an identifier which usually stands for number of elements to store in the data structure. In implementations, the second element is a valid asymptotic expression (see [15] for details in such valid expressions) using one or more of the identifiers introduced in the specification.

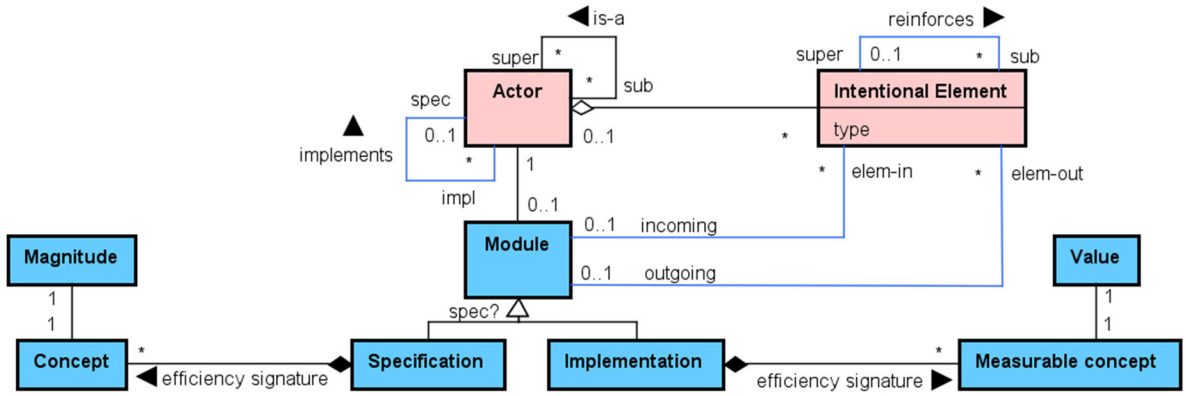


Figure 2: Extending the original iStar 2.0 metamodel (in salmon) to include new constructs (in blue).

Table 1 Summary of integrity constraints for the extended metamodel

ID	Integrity constraint
IC1	**context** IntentionalElement **inv**: incoming->size() + outgoing->size() = 1
IC2	**context** Module **inv**: spec? **implies** (not elem-in->exists(type=quality **or** type=task) and not elem-out->exists(type=quality **or** type=task))
IC3	**context** Actor **inv**: spec <> null **implies** module.spec? = false
IC4	**context** Actor **inv**: impl->notEmpty() **implies** spec.module.spec? = true
IC5	**context** IntentionalElement **inv**: super <> null **implies** (incoming.actor.super = super.incoming.actor or outgoing.actor.super = super.outgoing.actor)
IC6	**context** IntentionalElement **inv**: super <> null and type <> super.type **implies** ((type = task **or** type = resource) **implies** (super.type = goal **or** super.type = quality) **and** (type = goal **implies** super.type = quality))

4. Example: a hierarchy of data structures and their abstract data types

Figure 3 illustrates a hierarchy of abstract data types as they are presented in [18]. Due to the limited available space, we have neither included all possible abstract data types (e.g., we have not included trees) nor shown the complete contents of the modules, but we have selected some relevant dependencies and signature statements.

On the left-hand side, we find three main families of abstract data types: sequences, functions and graphs. Sequences support a general goal to their customers (among others), namely providing a concrete element stored therein (*Get*). The three specializations redefine this general goal in different ways (yellow dependum), e.g., in a stack, the element that is got is the one in the *Top*. These three specializations can be specialized themselves, and we present: *(i)* a particular type of *Queue*, namely *Priority Queue*, which requires the customer to specify what the *Priority* is and also redefines correspondingly the *Head* operation; *(ii)* an extension of *List* that supports traversals in both sequence's directions (*Bidirectional List*), adding a goal that allows to *Go Back* from the current one. *Functions* have been introduced in Section 2, and here we just remark the definition of *Sets* and going further, *Mathematical Sets*, adding some operation like *Union*. Concerning *Graphs*, the most general specification defines undirected and unlabeled graphs, and then two specializations separately add directed edges (by redefining the *Adj* goal into two direction-specific) and labels (by requiring the customer to define such concept), respectively. A last specialization *Labelled Graph* defines directed labeled graphs by inheriting from the two of them. This specialization is especially interesting since it supports some elaborated queries, as for instance the calculation of *Shortest Paths*.

On the right-hand side, we present a (incomplete) collection of implementations that suit these specifications. For *Sequences*, we exemplify with *Queues*. There are three strategies: *(i)* a *Sequential Queue* representation (all elements stored in consecutive positions of an array); *(ii)* a smarter specialization, namely *Circular Queue*, which avoids reallocations when the end of the array is reached; (iii) a *Linked Queue* representation allocating memory space as needed. The incoming/outgoing

dependencies reflect the characteristics of these implementations. We show also the *Heap* for the *Priority Queue* type of queue which demands to know the *Approximate Size* of the data structure. Similarly, for *Functions*, we show two general implementation strategies, namely using *Hash Tables* and *AVL Trees* (which allow to *Get Ordered List* of elements in the *Function*, provided that a *Greater Than* operation exists) and we refine the general concept of hash table into several strategies that differ in many respects, as illustrated in the dependencies that appear in the corresponding modules. Last, we present three main implementation strategies for graphs that apply to any graph variant, therefore we establish the *implements* link associating to the most general graph specification.

As an important remark, we would like to stress that this hierarchy is at the intentional level and not at the code level. This means that we should not think that every extended iStar 2.0 implementation module will yield to exactly one software component. The purpose of this hierarchy is not organizing code, but organizing a collection of abstract data types and data structures in a way that their similarities and differences at the intentional level become clearer.

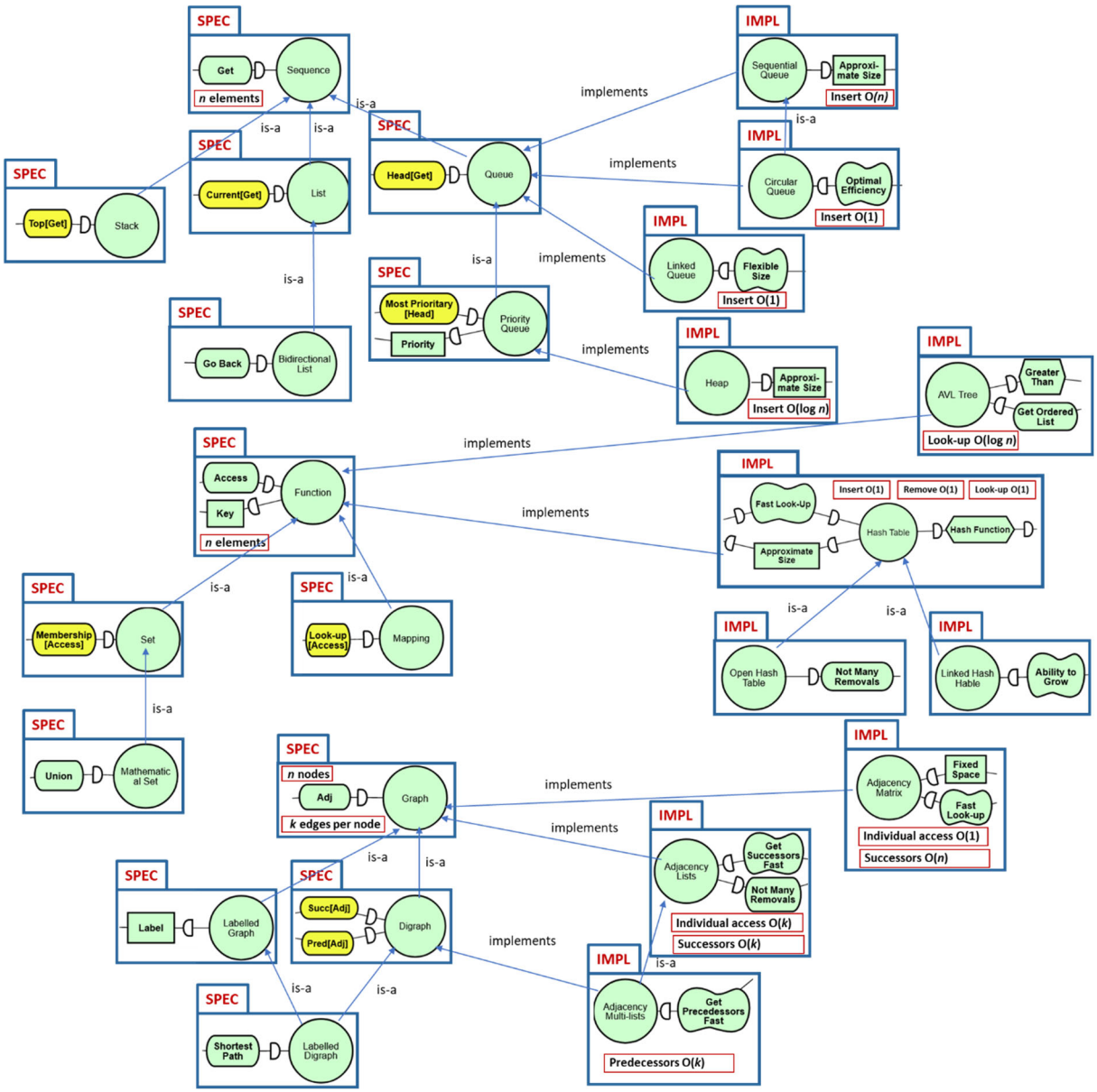


Figure 3: Abridged view of a hierarchy of abstract data types and data structures as presented in [18].

5. Conclusions and future work

In this paper we have continued our previous work on using *i** to describe data structures [1] in two different ways: *(i)* identifying and consolidating the constructs to be added onto iStar 2.0 for modeling in this particular domain; *(ii)* extending the iStar 2.0 metamodel [2] in order to support these constructs. To illustrate the proposal and to validate the adequacy of these constructs, we have modelled a hierarchy of data structures (and their corresponding specifications) that are usual to find in a typical data structure course. In fact, we foresee the context of teaching as the one that can benefit the most from our approach, and in this direction, we have conducted a first experiment evaluating our original proposal in a bachelor course on algorithms and data structures with promising results [19]. We plan to run a similar experiment with the new proposal presented in this paper.

There are other approaches that propose the use of modeling languages to describe abstract data types or data structures. For instance, Hoang et al. propose the use of class diagrams from iUML-B, an extension to UML using Event-B theories [20]. Contrary to our proposal, Hoang et al.'s work focus on validation issues and not in improving understandability and providing a unified view of the full spectrum of abstract data types and data structures.

As future research, we would like to apply the ideas presented in [10] to drive the selection of data structures in a particular context, with concrete functional and non-functional requirements. At the functional level, requirements can be compared against intentions appearing in the specification modules to find the best matches. In a subsequent step, in order to choose the best data structure, non-functional requirements can be compared against both the intentions and the efficiency signature. We also foresee the use of metrics over the resulting models [21][22] as a way to compute emerging properties of the solution, such as complexity of the solution (e.g., the more different data structures are combined, the more complex is the resulting program). Also, we want to face the challenge of integrating this graphical and intentional view provided by iStar 2.0 with the classical equational definition of abstract data structures. Furthermore, we want to add detail to the concept of *Measurable Concept* included in the metamodel as to capture complex conditions that may be required by modern uses of advanced data structures, e.g. memory usage or energy consumption for data structures in a mobile app development context [23]. Last, from a technical point of view, we want to implement the iStar 2.0 extension with the piStar tool[2] [24] that we have used in the paper for diagramming the iStar 2.0 actors and intentional elements.

References

[1] X. Franch, Using *i** to Describe Data Structures, in: Proceedings of the Thirteenth International iStar Workshop, iStar'20, CEUR Workshop Proceedings, 2641, 2020, pp. 49–54.

[2] F. Dalpiaz, X. Franch, J. Horkoff, iStar 2.0 Language Guide. arXiv preprint arXiv:1605.07767, 2016.

[3] X. Franch, Incorporating Modules into the *i** Framework, in: Proceedings of the 22nd International Conference on Advanced Information Systems Engineering, CAiSE'10, Lecture Notes in Computer Science, 6051, 2010, pp. 439–454.

[4] A. Maté, J. Trujillo, X. Franch, Adding Semantic Modules to improve Goal-oriented Analysis of Data Warehouses using I-star. Journal of Systems and Software 88 (2014): 102-111.

[5] L. López, X. Franch, J. Marco. Specialization in the iStar2.0 Language. IEEE Access 7 (2019): 146005-146023.

[6] M. Soares, J. Pimentel, J. Castro *et al.*, Automatic Generation of Architectural Models from Goal Models, in Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering, Knowledge Systems Institute, SEKE'12, 2012, pp. 444-447.

[7] I. Ayala, M. Amor, J.-M. Horcas, L. Fuentes, A Goal-driven Software Product Line Approach for Evolving Multi-agent Systems in the Internet of Things, Knowledge-Based Systems 184 (2019), 104883.

[2] https://www.cin.ufpe.br/~jhcp/pistar/

[8] A. Lavalle, A. Maté, J. Trujillo, S. Rizzi, Visualization Requirements for Business Intelligence Analytics: A Goal-Based, Iterative Framework, in: Proceedings of the 27th International Requirements Engineering Conference, RE'19, IEEE Computer Society, 2019, pp. 109-119.
[9] H. Estrada, A. Martínez, O. Pastor *et al.*, A Service-oriented Approach for the *i** Framework, in: Proceedings of the Third International iStar Workshop, iStar'08, CEUR Workshop Proceedings, 322, 2008, pp. 21–24.
[10] X. Franch, On the Lightweight Use of Goal-Oriented Models for Software Package Selection, in: Proceedings of the 17th International Conference on Advanced Information Systems Engineering, CAiSE'05, Lecture Notes in Computer Science, 3520, 2005, pp. 551-566.
[11] B.H. Liskov, J.V. Guttag, Abstraction and Specification in Program Development, MIT Press, 1986.
[12] E.S.K. Yu, Modelling Strategic Relationships for Process Reengineering, PhD. thesis, University of Toronto, 1995.
[13] D.E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms (3rd ed.), Addison Wesley Longman Publishing Co., Inc., 1997.
[14] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, E., Schonberg, Programming with Sets: An Introduction to SETL. Springer, 1986.
[15] P. Botella, X. Burgués, X. Franch, M. Huerta, G. Salázar, Modeling Non-Functional Requirements, in: Proceedings of Jornadas de Ingeniería de Requisitos Aplicada, JIRA 2001.
[16] D. Moody, The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering, Transactions on Software Engineering 35(6), 2009, pp. 756-779.
[17] J.J. Martin, Data Types and Data Structures, Prentice-Hall, 1986.
[18] X. Franch, Estructuras de Datos: Especificación, Diseño e Implementación. Editorial UPC, 1993.
[19] X. Franch, M. Ruíz, Goal-oriented Models for Teaching and Understanding Data Structures, in: Proceedings of the 40th International Conference on Conceptual Modeling, ER'21, Lecture Notes in Computer Science, 2021, in press.
[20] T.S. Hoang, C. Snook, D. Dghaym, M. Butler, Class-Diagrams for Abstract Data Types, in: D. Hung, D. Kapur (eds) Theoretical Aspects of Computing – ICTAC 2017, Lecture Notes in Computer Science, 10580, 2017, pp. 110–117.
[21] X. Franch, A Method for the Definition of Metrics over *i** Models, in: Proceedings of the 21st International Conference on Advanced Information Systems Engineering, CAiSE'09, Lecture Notes in Computer Science, 5565, 2009, pp. 201–215.
[22] X. Franch, G. Grau, C. Quer, A Framework for the Definition of Metrics for Actor-dependency Models, in: Proceedings of the 14th International Requirements Engineering Conference, RE'06, IEEE Computer Society, 2006, pp. 359–360.
[23] R. Saborido, R. Morales, F. Khomh *et al.*, Getting the most from Map Data Structures in Android, Empirical Software Engineering 23, 2018, pp. 2829–2864.
[24] J. Pimentel, J. Castro, piStar Tool – A Pluggable Online Tool for Goal Modeling, in: Proceedings of the 26th International Requirements Engineering Conference, RE'18, IEEE Computer Society, 2018, pp. 498–499.