Context-aware factors in rearchitecting two-level models into multilevel models

Mira Balaban, Igal Khitron and Azzam Maraee

Ben-Gurion University of the Negev, ISRAEL
{mira,khitron,mari}@cs.bgu.ac.il

Abstract. Multilevel Modeling (MLM) conceptualizes software models as layered architectures of sub-models that are inter-related by the instance-of relation. Multilevel rearchitecting has received considerable attention, but little attention has been given to the context of the transformed structures. In this paper we focus on possible contexts in MLM rearchitecture. We analyze factors of accidental complexity in MLM, suggest quantitative measures for these factors, and show how they can be used for evaluating alternative MLM transformations of two-level models. The context-aware factors are now being implemented in an FOML MLM tool.
Keywords: Multi-level modeling, context-aware multilevel architecture, accidental complexity factors, evaluation criteria.

1 Introduction

Multilevel modeling (MLM) is a software modeling approach in which a problem is represented using a sequence (or sequences) of modeling levels (also termed layers). Each level (but the lowest) is used for modeling its lower levels, or alternatively, elements in each level (but the highest) function as instances of elements in higher levels. Overall, the models in a multilevel representation are inter-related by the *instance-of* relation among objects and classes, and maybe further restricted by inter-level and intra-level constraints. The exact relationship between adjacent levels vary among approaches.

MLM started out of philosophical arguments relying on multilevel ontological classifications, where natural domains have multiple levels of classification. Along with the modeling arguments, there came pragmatic engineering arguments that point to reduced *accidental complexity* [1] when two level models of *type-instance* relationships are rearchitected using multilevel models [2–4]. Current studies of multilevel modeling concentrate on the rearchitecture of the types and instances under consideration, while mostly ignoring their context, i.e., the surrounding classes, associations and class hierarchy structures. An exception is [3].

In this paper we suggest general factors for evaluating the quality of multilevel models. We study a variety of *context-aware factors* like *redundancy, compositionality, coupling, cohesion* and *stability*, and suggest *quantitative measures* like *amount of duplication* and of *inter-level associations*. The factors are inter-related, sometimes overlapping and sometimes contradicting. The set of

measures can be used by a modeler to build his or her own subjective *quantitative scale* for measuring accidental complexity[1]. We apply the suggested scale for measuring accidental complexity of several options for context-aware multilevel rearchitecture. This scale is currently under implementation within the multilevel component of our FOML modeling application [7].

The contributions of this paper are: (1) a suggestion for a subjective quantifiable scale for measuring accidental complexity in MLM; (2) proposed patterns for context-aware multilevel rearchitecture of two-level models, with least accidental complexity. Section 2 presents the suggested quantitative scale for measuring accidental complexity in MLM, Section 3 discusses multilevel transformations, Section 4 summarizes related works, and Section 5 concludes the paper.

2 Factors and Quantitative Measures for Evaluating Accidental Complexity in MLM

The term *accidental complexity* was first used by Brooks in [1] as a way to capture non-essential complexity, caused by a language or a tool for implementation, and is not an essential characteristic of the solved problem. This concept has been used in the study of MLM for arguing that MLM reduces accidental complexity of modeling Type-Instance relationships [2,3]. However, when surrounding context in a model is considered, MLM rearchitecting might not be straightforward, and there are multiple alternatives, each with its own pros and cons.

We introduce *factors*, which are general features of models, and for each factor, we point to possible quantitative measures that can assign a numerical value to a given model. Factors and measures are motivated and demonstrated on an extended version of the Type-Instance pattern from [2]. The extended model, in Figure 1, adds a context of associated classes, and hierarchy of classes.[2]

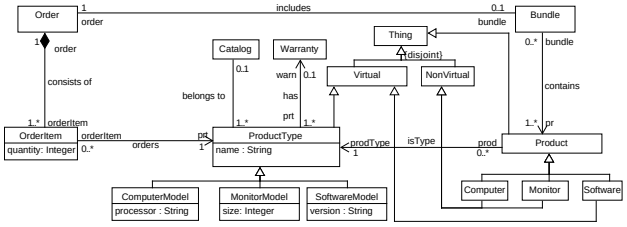

Fig. 1: The Type-Instance structure from [2] extended with context: Client, server and hierarchy classes

Redundancy

Redundancy can be measured by *duplication* of elements. This is the main accidental complexity argument in the Type-Instance pattern in [2]. The multilevel

[1] The analysis refers to the *potency*-based approach of [5,6].

[2] Part of this context appears in [2]

model is considerably smaller since instances of a type-class are combined with its instance-classes, into *clabjects*[3].

Duplication can arise in the multilevel rearchitecture of the context, as well. Figure 2 shows a possible three level model, for the type class *ProductType* and its context classes *OrderItem, Bundle*, from Figure 1. The multilevel model consists of two class models, at levels 2 and 1 (marked on the left corner as "@n"), and an instance model at level 0. The *potency-mechanism* controls the instantiation of classes and associations and of attribute inheritance between levels. A potency value n (visually marked by "@n" sign), restricts the number of successive instantiations to at most n. The default potency of an element is its level, and is not marked.

In Figure 2, class *OrderItem* is positioned on level 2, with potency value 2 (unlike in [3][4]), since *OrderItem* objects are needed in states on level 0. But if *OrderItem* is not sub-classified on level 1, then it is a singleton, which is duplicated on level 1, probably with four associations that instantiate association *orders* on level 2 (and must be constrained by XOR), as shown in Figure 2. Basically, every singleton with potency value > 1 causes duplication of the singleton class and its associations with the other classes, on the immediate lower level.

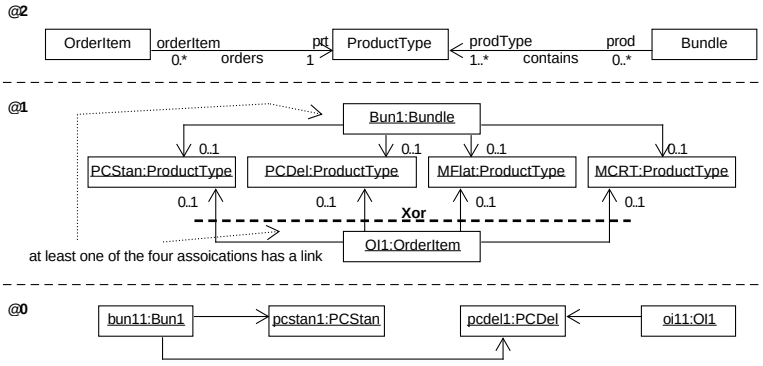

Fig. 2: Duplication and refinement: Singleton clients on level 2

Refinement

Refinement is a dual aspect to duplication, depending on context. An unnecessary duplication in one case, can be a desirable refinement in another. For example, if *Bundle*, like *OrderItem*, is positioned on level 2 with potency 2 and there are no domain specific bundle types on level 1, then it is a singleton, as in Figure 2. In that case, the *contains* association on level 2 is instantiated by four specific associations on level 1, with product specific multiplicity constraints.

[3] [3] presents a model in the Cloud domain, with a considerably less classes and objects in the multilevel rearchitecture of Type-Instance relationships.

[4] Figure 23 in [3].

That is, duplication turns into a desirable refinement rather than causing redundancy. Altogether, *duplication*, can measure accidental complexity factor in one case or a quality improvement factor, in another case.

Upward level-coupling: A level coupled with multiple higher levels
The pattern of de Lara et al. [3] suggests putting client classes of *Product* on level 2, with a *leap potency* value 2.[5] Following this pattern, *Bundle* and *contains* are instantiated directly on level 0, and *Bundle* objects on level 0 are linked to concrete monitors or computers. Since every bundle has an order, which has its order items, if class *Order* is placed on level 2 with potency 1 (like *OrderItem*), its instances reside on level 1, and bundles on level 0 are linked to orders on level 1, implying that a state on level 0 instantiates multiple classes in levels 1 and 2. A change in either of these levels might affect level 0. If *Order* on level 2 is given a leap potency 2 (following *Bundle*), then the order of a bundle resides on level 0, but its order items reside on level 1, implying again, dependency of level 0 in levels 1 and 2. These situations are shown in Figure 3 below.

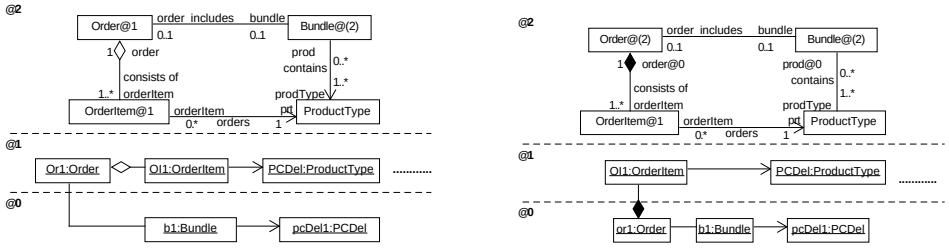

Fig. 3: Rearchitecture with an upward level coupling factor of accidental complexity

Altogether, *upward level-coupling* characterizes cases where a level might be affected by changes in multiple higher levels.[6] This factor can be caused by using leap potency for a class that is associated (directly or not) with a class with a smaller potency value. So, an association sequence between a class with leap potency value n and a class with potency $< n$ is a quantitative measure for this factor. Inter-level associations might also point to upward level-coupling.

Downward level-coupling: A level coupled with multiple lower levels
This factor is a dual of the previous one (reminiscent of the dual bad smell to Divergent Change[7]). In the situation described in Figure 3, elements in level 2 are instantiated on levels 1 and 0. Therefore, a change in level 2 might affect multiple

[5] *Leap potency*, marked with additional parentheses as in "@(n)", restricts the element to be instantiated only n levels below.
[6] Reminds the Divergent Change software bad smell.
[7] Shotgun Surgery is the bad smell characterizing software where a change in one place requires changes in multiple places in the code.

levels. A quantitative measure for this factor is the appearance of classes and associations with different potency values, including in particular, leap potency for classes. Inter-level associations might also point to a downward level-coupling.

Note that the upward and the downward level-coupling factors are not necessarily symmetric. If the problem is caused by a one direction inter-level association, it is possible that there is a coupling problem only in one direction.

Level instability

A major motivation for using MLM is to enable dynamic creation of new types, as shown in the multilevel rearchitecture of the static class hierarchy in [2]. This means that intermediate levels in a multilevel model can undergo changes, which usually are perceived as *model transformation*. Standard approaches in modeling distinguish between types and data, where types are stable – up to model transformation, while data is constantly changing. Consider Figures 3a and 3b. Since class *OrderItem* on level 2 has potency value 1, it is instantiated on level 1, which is a level of types, by objects that are not clabjects, and are dynamically changing, momentary states (snapshots). This implies constant instability to a level of types, which affects its lower instance levels: For every creation or deletion of a bundle on level 0 and its order, level 1 is modified.

Level instability is an accidental complexity factor that characterizes cases where a level of types is changed following state changes in a data level. This situation occurs when a class with potency value less than its level, is associated with a class C through an association a, where the potency (or the leap potency) value of C and a is their level.

Conceptualization

Conceptualization is a comparative factor, used to understand conceptual gains and loses in a rearchitecture transformation. For example, the concept of Type-Instance relationship is built into the multilevel model as a *first class citizen*, while not being supported in a two level model.

The potency-based approach has been shown to enable flexible control over *attribute inheritance*, at a level that is not supported by standard OO inheritance mechanisms. This feature extends also to *association inheritances*, as shown in [3]. In OO modeling, association inheritance can be constrained using the *redefinition* constraint. But in that case, the redefined association still uselessly exists for the subclass, causing duplication of associations. In MLM, the potency mechanism enables more flexible association inheritance.

In contrast to the above conceptual gains in MLM, splitting a class hierarchy between different levels of a multilevel model might cause lose of visibility for client classes, and might require additional operations. When breaking a class hierarchy structure, top classes in the lower levels lose their parent classes. Therefore, client classes need to duplicate their requests for all top classes, and might be affected by dynamic changes in lower levels of a class hierarchy.

Altogether, the multilevel rearchitecture might increase conceptualization in one direction, while reducing abstraction in another. For broken class hierarchy structures, the advantage of dynamic type creation and flexible inheritance should be balanced with the lose of abstraction in visibility.

Compositionality of a multilevel model

A multilevel model is composed of plain class models. Its semantics is composed from the semantics of its levels [8]. A correct multilevel model requires that every level is a class model, which is a partial instance of its immediate upper level, via a *mediator* that lists instantiation relationships. A legal instance of a multilevel model consists of instances of the level class models, that are required to extend the syntactic partial instance relationships. Therefore, a multilevel model without inter-level constraints is compositional. Compositionality is a desirable modeling feature since the semantics, management and understanding of a composite model can emerge from those of its components. It is closely related to the upward and downward coupling factors.

Compositionality is restricted by all kinds of cross layer constraints, including inter-level associations or links, leap-potency characterization, and explicit inter-level constraints [8]. A reasonable quantitative measure for compositionality is *number of inter-level constraints*.

Direct mapping to the denoted problem domain

This is a subjective factor, that might be related (or combined) with the *conceptualization* factor. *Direct mapping* refers to the ability to build a model that directly reflects and explicitly embeds the main intended abstractions. The multilevel architecture directly captures the basic distinction between types and their instances, which characterizes most modeling approaches. The level organization that is based on instantiation, enables level evolution that reflects dynamic creation of types in reality.

Level incohesion, caused by mixture of types and objects in the same level, breaks the type-instance distinction in a problem domain. For example, in the two versions of Figure 3, specific state-related *OrderItem* objects are linked to their product types, creating confusion that results from putting *OrderItem* on level 2, without having semantics of types of types. The potency 1 of *OrderItem*, creates class instances with potency 0 at level 1, which following [5], are objects and links. But since these objects are not stable static objects they create *level instability*, and cause other problems of coupling and compositionality. Level incohesion also reduce understandability, as discussed below.

Quantitative measures for direct mapping are subject to modeling ideals. If built-in separation of types and instances and multilevelling of typing are important abstractions, then clabject potency-values that violate the level ordering, is a measure for level incohesion (not including abstract classes). Using the leap-potency mechanism for clabjects also breaks the type multilevel hierarchy.

Understandability

Our cooperation with the developers of the Ink language [9] shows that working with MLM requires special training. A multilevel model is probably more difficult to understand than a standard two level model, that is structured by class hierarchies, associations and interfaces.

The clarity and understandability of a multilevel model are affected by syntactical features like the number of levels and number of constraints. An explicit

built-in visual constraint, e.g., a multiplicity constraint, is clearer than an implicit formulation written in an associated language, e.g., using the *size* OCL operation. Likewise, implicit inter-level constraints like inter-level associations or using leap-potency reduce understandability. Additional constraints, like the added XOR constraint in Figure 2, also reduce clarity. Quantitative measures for understandability might involve the number of levels, number of non-builtin constraints, number of inter-level constraints, counting additional constraints, and mixture of potency values in a level, which causes level incohesion.

3 Accidental Complexity of Context-Aware Multilevel Rearchitecture

In this section we present two alternative MLM transformations for the two level Type-Instance based model in Figure 1, and analyze and quantify their accidental complexity, using the measures from previous section.

De Lara et al. pattern [3]: The paper presents a pattern for MLM rearchitecting of client classes in a Type-Instance structure (Figure 23), which raises a number of concerns. Its application to Figure 1, is shown in Figure 4.

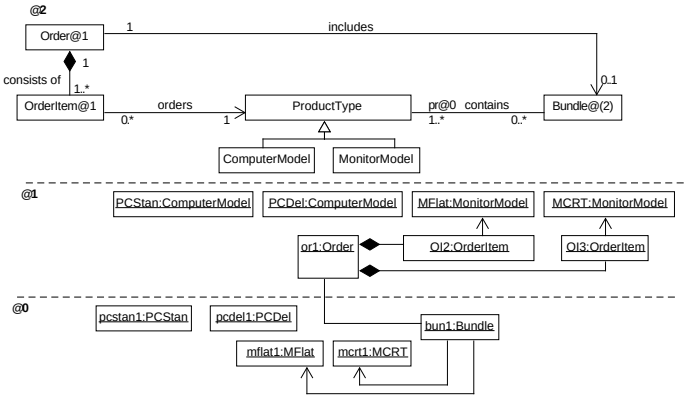

Fig. 4: Rearchitecting of Figure 1 following [3]

Figure 4 reveals several accidental complexity factors. The variety of potency values on level 2 points to a downward coupling problem; the multiple levels of instantiated classes of objects on level 0 points to an upward coupling problem; the leap potency and inter-level links point to a compositionality problem; the state-dependency status of *OrderItem* and *Order* objects points to level instability of level 1; the mixture of occasional inter-level links with stable types on level 1 points to level incohesion, which means lack of direct mapping to the denoted domain problem; the inter-level links and the leap-potency reduce understandability; the broken class-hierarchy of products, between levels 2 and 1 creates a visibility (conceptualization) problem. Altogether, we have counted 7 factors of accidental complexity, that result from context-aware considerations.

Level-aware multilevel rearchitecture of context classes: Context classes of a Type-Instance structure might be inter-related in complex and challenging ways, creating association cycles and tangled class-hierarchy structures. Therefore, it is likely that every multilevel rearchitecture of context classes reveals some accidental complexity factors, while avoiding others. In other words, there is no *silver bullet* transformation, that minimizes all accidental complexity factors. An advice for such transformations can declare ideals, and show reduced accidental complexity in the goal factors. The rearchitecture advice below emphasizes the *direct mapping to the problem domain* and *understandability* factors. The latter is essential in software modeling, and direct mapping increases understandability.

We suggest a context-aware architecture that determines the status of classes by comparative intended semantics, rather than by their context associations. According to this approach, the level of a class is determined by the question: "What is its semantically intended level?".

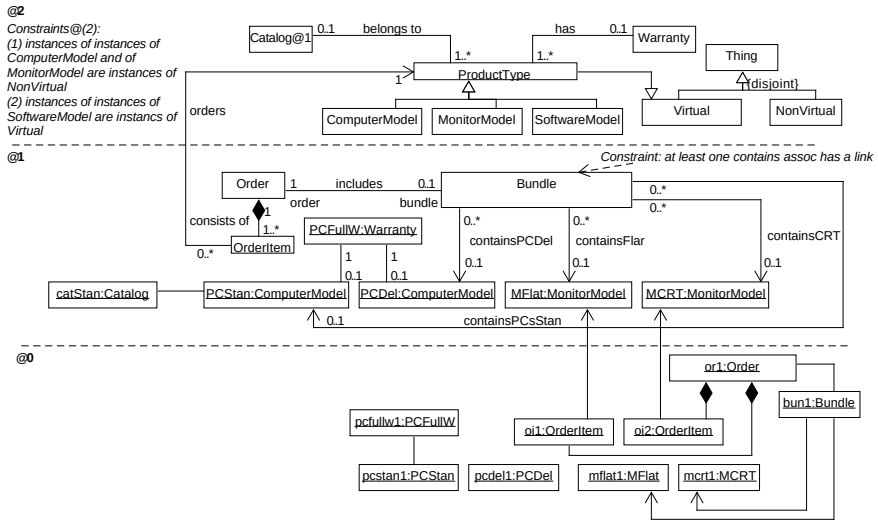

Fig. 5: Context-aware multilevel rearchitecting of Figure 1

Classes *OrderItem, Order, Bundle*, in Figure 5, are positioned on level 1, since they have an intended semantics of types. This leads to an inter-level *orders* association between *OrderItem* and *ProductType*, which reduces compositionality and understandability, but increases visibility, as an *Order* object on level 0 can ask its order-items for their product types, without listing specific types.

The multilevel positioning of the *Catalog* and the *Warranty* classes is also determined by the intended type semantics: A *Warranty* object is associated with every *bundle* object, meaning that there should be warranties on level 0. These warranties are of a variety of types, and therefore, *Warranty* is positioned on level 2 with potency 2. *Catalog*, on the other hand, is only associated with types of products, and not related to concrete products. Therefore, it is positioned on

level 2, with potency 1, and its instance objects on level 1 are linked to product types. This decision creates mixture of object and types, but as catalogs are not linked to objects on level 0, there is no level instability and level incohesion. Due to space limitations we avoid discussion of rearchitecting class-hierarchy context structures (e.g., classes *Virtual, nonVirtual* in Figure 1).

Altogether, the suggested context-aware multilevel rearchitecture causes reduced compositionality and understandability, but increases visibility. Compared with the 7 factors in the rearchitecture in Figure 4 this is, a meaningful improvement.

Towards automating context-aware multilevel rearchitecture: The analysis of accidental complexity factors and the above examples show that there is no single best transformation. Automation depends on one's values and ideals. Following our advice for rearchitecting the complex context in Figure 1, we suggest transformation rules, organized by priorities.[8]

1. **Direct or indirect associated classes of the *Product* class, in a Type-Instance structure:** Such classes join the *Product* class, in the same level, preserving the association structure, like class *OrderItem, Order* and *Bundle* in Figure 5. If there is an association cycle between the *Product* and the *ProductType* classes, this advice creates an inter-level association.
2. **Direct or indirect associated classes of the *ProductType* class, in a Type-Instance structure:** Such classes, like *Warranty* and *Catalog* in Figure 5, join the *ProductType* class, in the same level, provided that they are not positioned in a different level, following the advice of previous entry.
3. **Hierarchy related classes of transformed classes – sibling classes:** Sibling classes should better reside on the same level in the multilevel architecture. Together with the previous advice, it means that sibling classes of say, *Order* or *Bundle* are advised to be classified together, on the same level.
4. **Hierarchy related classes of transformed classes – ancestor or descendant classes:** In principle, ancestor and descendant classes should go together. In cases of deep class hierarchy structures, or if hierarchy structures include classes that are already classified on different levels, more heuristic advice is needed, to balance accidental complexity with modeling ideals.

4 Related Work

Quality of models has been evaluated by a variety of metrics, such as number of associations, aggregations, generalizations and dependencies [10, 11]. Model metrics are used for identifying modeling problems and predicting maintainability, understandability and reusability. Stability of models is evaluated by metrics that rely on the amount of changes, relatively to the initial version [12].

The notion of *Accidental complexity* was introduced in [1] in order to distinguish between *essential complexity*, which characterizes a problem domain,

[8] This is not refactoring since replacing subtyping or association by type membership changes the semantics.

to accidental one, that results from using weak implementation languages, weak abstractions, or bad design. This notion is frequently used in companion with a variety of software metrics, and also with respect to efficiency of concrete applications [13]. In MLM, transformations from two level modeling reduce redundancy and improve conceptuaization, but impact on context classes was not investigated.

5 Conclusion and Future Work

In this paper we have analyzed factors of accidental complexity in MLM, and show that they might arise in MLM when context elements are considered, We provided quantitative measures for these factors, and used them to compare alternative multilevel transformations. Finally, we provide advice for (semi)-automatic, context aware multilevel transformation. The factors suggested in the paper are now being implemented in our FOML-based MLM tool [7, 14]. In the future we plan on developing a semi-automated MLM transformation tool.

References

1. Brooks, F.P.: No silver bullet – essence and accident in software engineering. IEEE Computer **20** (1987) 10–19
2. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software & Systems Modeling **7** (2008) 345–359
3. de Lara, J., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Trans. Softw. Eng. Methodol. **24** (2014) 12:1–12:46
4. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific meta-modelling languages. SoSyM **14** (2013) 429–459
5. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: UML. Springer (2001) 19–33
6. Atkinson, C., Kühne, T.: Rearchitecting the uml infrastructure. ACM Trans. Model. Comput. Simul. **12** (2002) 290–321
7. Khitron, I., Balaban, M., Kifer, M.: The FOML Site. `https://goo.gl/AgxmMc` (2017)
8. Balaban, M., Khitron, I., Kifer, M., Maraee, A.: Formal executable theory of multilevel modeling. In: CAISE. (2018)
9. Acherkan, E., Hen-Tov, A., Lorenz, D.H., Schachter, L.: The ink language meta-metamodel for adaptive object-model frameworks. In: ACM Int. Conf. Comp. on OO Prog. Sys. Lang. and Appl. Companion. OOPSLA, 181–182 (2011)
10. Genero, M., Manso, E., Visaggio, A., Canfora, G., Piattini, M.: Building measure-based prediction models for uml class diagram maintainability. Empirical Software Engineering **12** (2007) 517–549
11. Genero, M., Piattini, M., Calero, C.: A survey of metrics for uml class diagrams. Journal of object technology **4** (2005) 59–92
12. AbuHassan, A., Alshayeb, M.: A metrics suite for uml model stability. Software & Systems Modeling (2016) 1–27
13. Haslum, P., et al.: Reducing accidental complexity in planning problems. In: IJCAI. (2007) 1898–1903
14. Balaban, M., Kifer, M.: Logic-Based Model-Level Software Development with F-OML. In: MoDELS 2011. (2011)