

On Structured Model-Driven Transformations

Roberto Bruni¹, Alberto Lluch Lafuente², and Ugo Montanari¹

¹ (Department of Computer Science, University of Pisa, Italy)

² (Computer Science and Applications, IMT Lucca, Italy)

Abstract Structural aspects play a key role in the model-driven development of software systems. Effective techniques and tools must therefore be based on suitable representation formalisms that facilitate the specification, manipulation and analysis of the structure of models. Graphical and algebraic approaches have been shown to be very successful for such purposes: 1) graphs offer natural a representation of topological structures, 2) algebras offer a natural representation of compositional structures, 3) both graphs and algebras can be manipulated in a declarative way by means of rule-based techniques, 4) they allow for a layered presentation of models that enables compositional techniques and favours scalability. Most of the existing approaches represent such layering in a plain manner by overlapping the intra- and the inter-layered structure. It has been shown that some layering structures can be conveniently represented by an explicit hierarchical structure enabling then structurally inductive manipulations of the resulting models. Moreover, providing an inductive presentation of the structure facilitates the compositional analysis and verification of models. In this paper we compare and reconcile some recent approaches and synthesise them into an algebraic and graph-based formalism for representing and manipulating models with inductively defined hierarchical structure.

Key words: hierarchical graphs; rewriting logic; model transformations

Bruni R, Lluch Lafuente A, Montanari U. On structured model-driven transformations.

Int J Software Informatics, Vol.5, No.1-2 (2011), Part I: 185–206. <http://www.ijsi.org/1673-7288/5/i89.htm>

1 Introduction

Model-Driven development is a software engineering paradigm built around the notion of *model*. Models are abstract descriptions of software artifacts that facilitate understanding, master complexity and therefore have a positive impact on the productivity, quality and cost of software development. Among the different features of a software artifact, *structure* is a key one. Think for instance about the different styles in which components in a software architecture can be linked together or organised (star, ring, tiers, etc.) or the interrelations in a class diagram. Model-driven development requires effective techniques and tools based on suitable formalisms for the specification and manipulation of the structure of models. Graphical and algebraic approaches have both been shown to be very successful for such purposes: 1) graphs offer a visual representation of the topological structure of a model, 2) algebras offer

This work is sponsored by the EU Project ASCENS and the Italian MIUR Project IPODS.

Corresponding author: Alberto Lluch Lafuente, Email: alberto.lluch@imtlucca.it

Received 2010-10-07; Accepted 2011-01-03; Final revised version 2011-01-17.

a natural representation of the compositional structure, 3) both graphs and algebras can be manipulated in a declarative way by means of rule-based techniques, 4) they allow for a layered presentation of models that enables compositional techniques and favours scalability. In this paper we summarise and reconcile some recent work^[1-2,7] proposing an algebraic and graph-based formalism for representing and manipulating models with hierarchical structure. We believe that our presentation provides evidence that it is possible to exploit the structure of models to enhance software description and to facilitate model transformations.

Many domains exhibit an inherently hierarchical structure that can be exploited conveniently. We mention, among others, nested components in software architectures, nested sessions and transactions in business processes, nested membranes in computational biology, and so on. A prominent example can be found in the Meta-Object Facility¹⁾ (MOF), a de-facto standard in model-driven engineering. The MOF defines a meta-modelling paradigm by providing a set of UML-like structural modelling primitives including *composition* associations. Such associations are required to satisfy certain constraints, like unique and acyclic containment, that establish a hierarchical structure on models. However, very often such layering is represented in a plain manner by overlapping the intra- and the inter-layered structure. For instance, models are usually formalised as flat configurations (e.g. graphs) and their manipulation is studied with tools and techniques based on term rewriting^[20] or graph transformation theories^[12,22] that do not exploit the hierarchical structure. In the case of MOF, models are collections of objects that may refer to other objects through references, corresponding to flat graphs in the traditional sense. Some of these references are typed as composition associations, whose semantics corresponds to structural containment. In this way, models have an *implicit* nested structure since some objects may contain other objects, i.e. containment references are defined as distinguished edges in a graph, receiving a particular treatment during graph transformations^[3,7]. On the other hand, an *explicit* treatment of structural containment for specifying and transforming model-based software artifacts is possible. As a matter of fact, some layering structures (like composition relations in MOF) can be conveniently represented by an explicit hierarchical structure enabling then hierarchical manipulations of the resulting models (see e.g. Refs.[1, 5]).

Manfred Broy research involved hierarchical approaches in software engineering with a particular attention to software models, see e.g. Refs.[8-10, 17, 18] to mention a few recent contributions. In Refs.[17, 18] Manfred introduced a novel extension to UML sequence diagrams to describe interaction scenarios in hierarchical broadcasting architectures, and presented a methodology that exploits those scenarios in order to derive structural and behavioural aspects of the architecture under development. In Ref. [8] Manfred studied different models of distributed, embedded software systems focusing on the various features of such systems such as data, states, interfaces, functionality, processes and, more relevantly, hierarchical components. The work introduces a mathematical model that included a means for representing abstraction and refinement relations as well as forms of composition and modularity. In Ref. [9] Manfred introduced a theory for modelling software intensive systems. He considered two orthogonal views: the task view, which regards the hierarchy of functions, fea-

¹⁾ <http://www.omg.org/mof/>

tures and services provided by the system, and the architectural view, which focuses on the components that form the system and their cooperation through behavioural interfaces. In Ref. [10] Manfred discusses the role of rigorous model development in software engineering, highlighting its success “on creating models or abstractions, more close to some particular domain concepts rather than programming, computing and algorithmic concepts” in contrast to traditional use of formal methods, and promoting their seamless adoption.

This paper honours Manfred Broy by considering the issue of hierarchical modelling of software artifacts, and by borrowing the *production plant* scenario used in some of his papers (namely^[17,18]).

A key instrument to discipline model-driven engineering is the concept of *meta-model*, i.e. a sort of meta-language describing the syntax of models that allows one, for instance, to restrict the admissible structures and patterns to be considered. Meta-modelling the structure of software artifacts involves declaring the classes, attributes and relations between the different entities. Types, grammars, algebras, logic and other basic principles have been combined into suitable meta-modelling mechanisms since the beginning of this discipline.

For instance, UML class diagrams use types to declare basic entity classes and logical constraints to restrict their interrelations. Many formalisations have been presented for such approaches, notably the algebraic semantics of the MOF^[7]. On the other hand, grammar based approaches were initiated by Lé Metayer in the field of programming languages^[14] and further exported to architectural styles using graph grammars^[15–16,19]. This tradition has given birth to an algebraic approach based on conditional term rewriting called ADR^[6] that has turned out to be very expressive and flexible. The most relevant application of ADR is for the modelling and analysis of SOA architectures with UML^[2].

Many engineering activities are devoted to manipulate software artifacts or to declare their dynamics. This issue is usually known as *model transformation*. For what regards the structure of models, transformations are defined at any level of a meta-modelling hierarchy: e.g. a transformation might define an architectural reconfiguration, a software refactoring or a language translation. Rule-based specifications have been very successful as a declarative approach in model-driven transformations. One of the key success factors are the solid foundations offered by rule-based machineries like term and graph rewriting. Still, the complexity of realistic problems requires suitable techniques to guarantee the scalability of rule-based approaches. We explain in this paper how to exploit the hierarchical structure of models for such purpose. The main idea is that *structured* models represented by terms are those that can be therefore manipulated by means of term-rewrite techniques. In particular one can use *conditional* term rewrite rules à la Meseguer^[20]. To coordinate and guide the manipulation of models, one can declare rules in the style of *Structural Operational Semantics*^[10] (SOS) and its implementation in rewriting logic^[23].

One interesting outcome of our work is the analysis and comparison of two different approaches that poses the basis for their combined, synergic use. The first approach (in the style of Ref. [7]) imposes a membership mechanism to classify object collections as meta-model conformant or not. The second approach (in the style of Ref. [2]), instead, tries to provide directly a signature of meta-model conformant

configurations. In the end we promote the conjoint use of both approaches: from the first approach we take the model based on collections of interrelated objects; from the second one we inherit the explicit treatment of structural containment.

Synopsis. Section 2 fixes the main notation we shall use in the rest of the paper. Section 4 describes our running example, borrowed from Manfred Broy's works. Section 3 presents a graph-based algebraic representation of models as nested object collections that makes an explicit treatment of hierarchical structures. Section 4 overviews two approaches for meta-modelling and suggests how to conciliate them. Section 6 explains rewrite rule formats for defining model transformations. Section 7 overviews some experimental results regarding the performance of the explained styles when actually executing model transformations. Section 8 concludes the paper.

2 Preliminaries: Rewriting Logic and Maude

Our machinery heavily relies on rewriting logic^[20] due to its well-developed theory, expressiveness, flexibility, generality and tool support. In particular, we will see that our models are terms of a particular signature of object configurations. This concept is crucial for understanding our work. Indeed, as we shall see in the rest of the paper, meta-models become just specialisations of that signature, while model transformations are programmed as rewrite theories over them.

A *rewrite theory* \mathcal{R} is a tuple $\langle \Sigma, E, R \rangle$ where Σ is a signature, specifying the basic syntax (function symbols) and type machinery (sorts, kinds and subsorting) for terms, e.g. model descriptions; E is a set of (possibly conditional) equations, which induce equivalence classes of terms, and (possibly conditional) membership predicates, which refine the typing information; R is a set of (possibly conditional) rules, e.g. model transformations.

The Maude framework^[11] provides a language for describing such rewrite theories and a tool built upon a rewrite engine for executing and analysing them. In the rest of the paper we shall use Maude's syntax, introducing the syntactic ingredients as we use them.

The signature Σ and the equations E of a rewrite theory form a *membership equational theory* $\langle \Sigma, E \rangle$, whose initial algebra is denoted by $T_{\Sigma/E}$. Indeed, $T_{\Sigma/E}$ is the state space of a rewrite theory, i.e. states (e.g. models) are equivalence classes of Σ -terms modulo the least congruence induced by the axioms in E (denoted by $[t]_E$ or t for short). Sort declarations takes the form `sort S` and subsorting is written `subsort S < T`. For example, we can declare sorts `sort Qid` for *quoted identifiers* (which Maude has built-in) and `sort QidList` for lists of identifiers and then allow a single identifier to be seen as a list by declaring `subsort Qid < QidList`. Operators are declared in Maude notation as `op f : TL -> T [a]` where f is the operator symbol (possibly with mixfix notation where underscores stands for argument placeholders), TL is a (possibly empty, blank separated) list of domain sorts, T is the sort of the co-domain, and a is a set of equational attributes (e.g. associativity, commutativity). For example, `op nil : -> QidList` declares a constant of sort `QidList` (for the empty list), `op _ _ : QidList QidList -> QidList [assoc id: nil]` declares juxtaposition of lists (i.e. *concatenation*) as an associative operator with unit `nil`. We shall present a signature Σ containing sorts and operators for describing models as collections of attributed, interrelated objects.

Equations that cannot be declared as equational attributes must be treated as functions defined by a set of confluent and terminating (possibly conditional) equations of the form $\text{ceq } t = t' \text{ if } c$, where t, t' are Σ -terms, and c is an application condition. When the application condition is vacuous, the simpler syntax $\text{eq } t = t'$ can be used. For example, assuming the length operator has been defined as $\text{op len} : \text{QidList} \rightarrow \text{Nat}$, then we can write its inductively defining equations as $\text{eq len}(\text{nil}) = 0$ and $\text{eq len}(I L) = 1 + \text{len}(L)$, where I is a variable of sort Qid and L is a variable of sort QidList . Roughly, an equational rule can be applied to a term t'' if we find a match for t at some place in t'' such that c holds (after the application of the substitution induced by the match). The effect is that of substituting the matched part with t' (after the application of the substitution induced by the match). For example, the term $\text{len}('a 'b 'c)$ reduces to $1 + \text{len}('b 'c)$, then to $2 + \text{len}('c)$ and finally to 3. In our theories, object collections are seen as multisets of objects, i.e. modulo the equational attributes for associativity, commutativity, and identity, thus axiomatising the graph-theoretic nature of models.

A membership predicate has the form $\text{cmb } t : T \text{ if } c$, where t is a Σ -term of some supersort T' of T and c is a predicate over t conditioning the membership statement. When the application condition is vacuous, the simpler syntax $\text{mb } t : T$ can be used. Roughly, a membership predicate states that if we are able to match a term t' with t such that c holds (after the application of the substitution induced by the match), then t' has sort T . For example, the previous subsort declaration can be written as the membership predicate $\text{cmb } I : \text{QidList} \text{ if } I : \text{Qid}$. Membership predicates provide a subtyping mechanism that we can use for defining meta-model conformance.

Rewrite rules are of the form $\text{cr1 } t \Rightarrow t' \text{ if } c$, where t, t' are Σ -terms, and c is an application condition (a predicate on the terms involved in the rewrite, further rewrites whose result can be reused, membership predicates, etc.). When the application condition is vacuous, the simpler syntax $\text{r1 } t \Rightarrow t'$ can be used. For example, the rule $\text{r1 } I L \Rightarrow L I \text{ if } L \neq \text{nil}$ is a rule for rotating the elements of a list. Matching and rule application are similar to the case of equations with the main difference being that rules are not required to be confluent and terminating (as they represent possibly non-deterministic concurrent actions rather than functions). For example, the term $'a 'b 'c$ can rewrite to $'b 'c 'a$ or to $'b 'a 'c$ or to $'a 'c 'b$, depending where the rule is applied. Equational simplification has precedence over rule application in order to simulate rule application modulo equational equivalence. Rewrite rules can be used to program model transformations in a declarative way.

3 Running Example: Production Plant

We consider a running example inspired by the *Product Automation* case study from Refs.[17,18]. We borrow the main ingredients of the example and adapt them for the sake of a better illustration of our concerns. The system under study is an autonomous transport system within a production plant. The production plant consists of workpieces that must be processed by machine tools. Workpieces might be subject to different processes carried out by machine tools. Both machine tools and workpieces are distributed among the various locations of the plant, which requires the system to transport pieces to the location of the machine tools that must operate

on them. Transportation is carried out by autonomous vehicles that negotiate with machine tools where to deliver the workpieces. Such negotiation is supported by a (hierarchical) broadcast system. For the sake of simplicity, we will focus on the spatial distribution and transportation of machine tools and workpieces and we will abstract away from vehicles and other entities involved in the actual case study. Figure 1 illustrates a configuration of the scenario in an informal way. The production plant is physically divided into rooms or areas denoted by rounded boxes. Machine tools are represented by gears, and workpieces by cylinders. Arrows are used to link workpieces to the machine tool they are assigned to for the next operation.

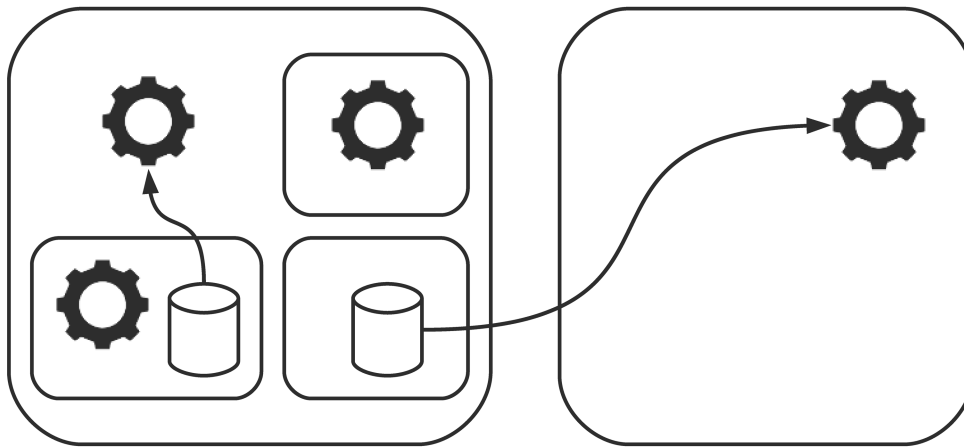


Figure 1. Production plant scenario

It is worth to remark that the aim of this example is purely illustrative. It focuses on the architectural view of a model, but our approach, of course, deals with models in the general sense.

4 Models

This section gives a formal representation of models based on an algebra of *nested object collections*, which can be understood as a particular class of attributed, hierarchical graphs. First we summarize an algebra of *object collections* that is used to represent models as flat graphs (see Fig.2). Next, we introduce *nested object collections* which allows us to see models as hierarchical graphs (see Fig.3).

4.1 Models as attributed graphs

In our setting a model is a collection of attributed objects. Maude already provides a signature for this purpose, called object-based signature^[11], which we tend to follow with slight modifications aimed to ease the presentation. Each object represents an entity and its properties. Technically, an object is defined by its identifier (of sort *Oid*), its class (of sort *Cid*) and its attributes (of sort *AttSet*). Objects are build with an operation $\langle _ : _ | _ \rangle$ with functional type $\text{Oid Cid AttSet} \rightarrow \text{Obj}$. Following Maude conventions, we shall use quoted identifiers as object identifiers. Class identifiers, instead will be defined by ad-hoc constructors. For instance

in our running example we use the constants `Room`, `Tool` and `Piece` of sort `Cid` to denote the classes of rooms, machine tools and workpieces, respectively.

The attributes of an object define its properties and relations to other objects. They are basically of two kinds: datatype attributes and association ends. Datatype attributes take the form `n: v`, where `n` is the attribute name and `v` is the attribute value. For instance, in our running example we shall consider an attribute `alarm` with domain in `{on,off}` (sort `Alarm`), representing whether the room alarm is activated or not. Similarly, we will consider an attribute `state` for denoting whether a machine is `ready` or `busy` (sort `State`), and an attribute `phase` for the completion phase (here a natural number of sort `Nat`, for simplicity) of a workpiece.

As an example of an attributed object, consider Fig.2 whose format is reminiscent of the UML notation, with boxes representing objects where the top frame contains the object identifier and its class and the bottom frame is reserved for datatype attributes. The room `'store2` on the right of Fig.2 is denoted in Maude syntax with `< 'store2 : Room | alarm: off >`.

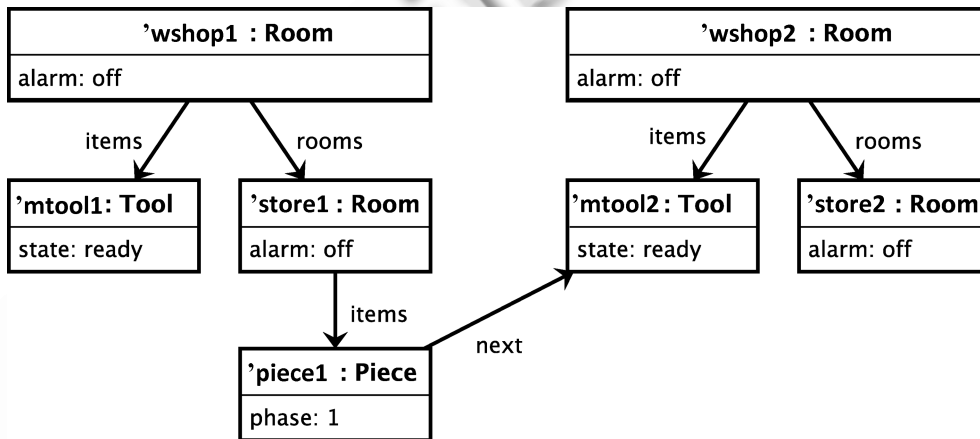


Figure 2. Production plant scenario in flat representation

In a configuration, objects are interrelated. Relations between objects can be represented in different ways. One typical approach is to use a pair of references (called *association ends* in UML terminology) for each relation. So if an object `o1` is in relation `R` with object `o2` then `o1` will be equipped with a reference to `o2` and vice versa. In our case this is achieved with attributes of the form `ends(R): O2` and `opp(R): O1` where `R` indicates the relation name and `O1`, `O2` are sets of object identifiers (sort `OidSet`). Association ends of the same relation within one object are grouped together (hence the use of identifier sets as domain of association attributes). Consider the location containment relation `rooms`, used to represent the fact that a large room can contain smaller ones. Room `'wshop2` contains the above mentioned room `'store2`. Both objects are hence denoted with `< 'wshop2 : Room | alarm: off , ends(rooms): 'store2 >` and `< 'store2 : Room | alarm: off , opp(rooms): 'wshop2 >`, respectively. Note that each association pair is graphically denoted with an arrow in Fig.2, which goes from the first object of the relation tuple to the second one.

Of course an object can be equipped with any number of attributes. Actually,

the attributes of an object form a set built out of singleton attributes, the empty set (`none`) and union set (denoted with `_ , _`).

Object configurations are essentially sets of objects. The sort for configurations is called `Conf` and its constructors are the empty configuration (`none`), singleton objects (as `Obj` is declared a subsort of `Conf`) and set union (denoted with juxtaposition).

As an example the whole configuration of Fig.2 is denoted with

```
< 'wshop1 : Room | alarm: off , ends(rooms): 'store1 , ends(items): 'mtool1 >
< 'mtool1 : Tool | state: ready , opp(items): 'wshop1 >
< 'store1 : Room | alarm: off , opp(rooms): 'wshop1 , ends(items): 'piece1 >
< 'piece1 : Piece | phase: 1 , opp(items): 'store1 , ends(next): 'mtool2 >
< 'wshop2 : Room | alarm: off , ends(rooms): 'store2 , ends(items): 'mtool2 >
< 'mtool2 : Tool | state: ready , opp(items): 'wshop2 , opp(next): 'piece1 >
< 'store2 : Room | alarm: off , opp(rooms): 'wshop2 >
```

In order to distinguish a model from the collection of objects that forms it, we wrap configurations together into a model with operation `<< _ >> : Conf -> Model`.

4.2 From flat to nested collections (and back)

The key observation now is that a prominent type of relation among objects is *structural containment* (e.g. UML composition associations). To capture such concept in an explicit way, we introduce the notion of *nested object collections*, taken from Ref. [1]. A *nested object collection* is realised using a very similar notation to that for ordinary, plain collections. The idea is as simple as to allow object collections to be the domain of attributes. We denote this third class of attributes as *container attributes*. Hence, the idea is that while in a plain object collection a containment relation `r` between two objects `o1` and `o2` is represented by attributing them with a pair of association end attributes `ends(r)` and `opp(r)`, now `o2` is embedded into `o1` by means of the container attribute `cont(r)`. For instance, the example configuration of our running example is now denoted with:

```
< 'wshop1 : Room | alarm: off ,
  cont(items): < 'mtool1 : Tool | state: ready > ,
  cont(rooms): < 'store1 : Room | alarm: off ,
    cont(items): < 'piece1 : Piece | phase: 1 ,
      ends(next): 'mtool2 >>>
< 'wshop2 : Room | alarm: off ,
  cont(items): < 'mtool2 : Tool | state: ready , opp(next): 'piece1 >
  cont(rooms): < 'store2 : Room | alarm: off >>
```

A graphical illustration that makes the embedding explicit is in Fig.3. We remark that the point is not in the visualisation of models. Indeed, our choice of visualising flat configurations in a tree-like manner using an ordinary plain graph (Fig.2) and nested configurations in a boxes-inside-boxes manner using a hierarchical graph (Fig.3) is arbitrary. The point is instead in the formal notation: nested configurations retain structural containment in the syntactic structure of terms.

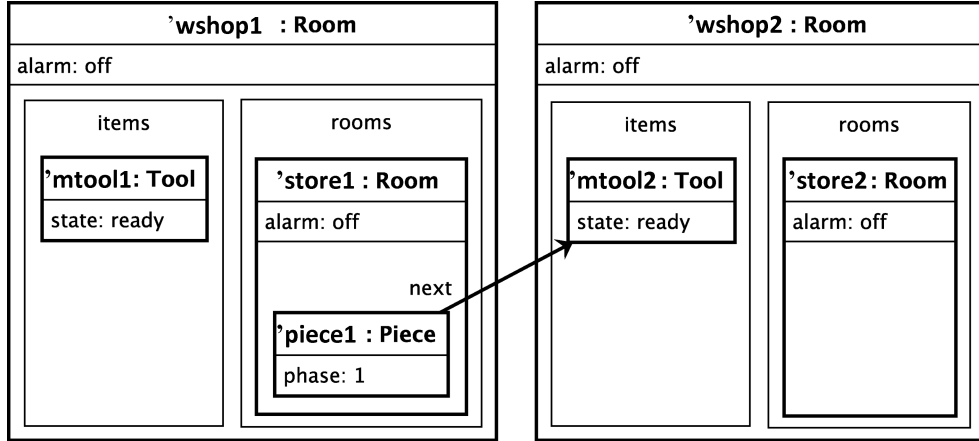


Figure 3. Production plan scenario in nested representation

The hierarchical structure of models forms a tree. The two approaches that we have described differ essentially in the way we represent such a tree. It is not by accident that each of the representations coincides with one of the alternative definition of trees offered by computer scientists: as a particular class of graphs (acyclic and connected) or as an inductively defined mathematical structure (a tree is a leaf, or a node with some subtrees). The literature provides enough examples where one representation is preferred over the other. Obviously, the main virtues and drawbacks of each representation style have their major impact when it comes to definitions, analysis, and manipulations. We postpone hence, the main discussion to the next sections and refer to Ref. [4] for a generic discussion and pointers about the benefits of expliciting the structure of graphs in software modelling.

Interestingly, flat and nested representations are in bijective correspondence, i.e., for each flat object collection we can obtain a unique nested collection and vice versa. To make this more explicit we outline such transformations below. We first define the function $||_||$ that allows us to *flatten* a nested collection by recursively replacing embedded objects by association ends (cf. the third equation).

```

eq || objCol1 objCol2 || = || objCol1 || || objCol2 ||
  if objCol1 /= none /\ objCol2 /= none .

eq || < oid1 : cid1 | attSet1 , cont(containerId1) : none > || =
  || < oid1 : cid1 | attSet1 > || .

eq || < oid1 : cid1 | attSet1 ,
      cont(containerId1) : objCol1 < oid2 : cid2 | attSet2 > > || =
  || < oid1 : cid1 | attSet1 ,
      cont(containerId1) : objCol1 ,
      ends(containerId1) : oid2 > ||
  || < oid2 : cid2 | attSet2 , opp(containerId1) : oid1 > || .

eq || obj1 || = obj1 [owise] .

```

The inverse function $[[_]]$ (from flat collections to nested ones) is defined below:

```

eq [[ < oid1 : cid1 | attSet1 , ends(containerId1) : none > objCol2 ]] =

```

```

[[ < oid1 : cid1 | attSet1 , cont(containerId1) : none > objCol2 ]] .

ceq [[< oid1:cid1 | attSet1 , ends(containerId1) : (oid2 oidSet1) >
  < oid2 : cid2 | attSet2 , opp(containerId1) : oid1 >
  objCol2 ]] =
  [[ < oid1 : cid1 | attSet1 ,
    ends(containerId1) : oidSet1 ,
    cont(containerId1) : < oid2 : cid2 | attSet2 > >
  objCol2 ]]
  if noCompositionLinkIn(attSet2) .

eq [[ objCol1 ]] = objCol1 [owise] .

```

where `noCompositionLinkIn` is a boolean function that checks whether a configuration is free of composition association links. Indeed, this property is exploited to ensure that each step of the transformation embeds an already nested object, i.e. it forces a bottom-up transformation based on the tree induced by composition association links.

Clearly, both the flattening and the nesting functions are bijective and indeed one the inverse of the other. This means that both flat and hierarchical representations are somehow isomorphic and we can pass from one to the other as we find more convenient for specific applications or analyses.

5 Meta-Models

We present in this section two approaches to the formalisation of the concepts of *meta-model* and *meta-model conformance* and discuss the way each approach deals with structural containment. We first explain what we intend under *meta-model*; roughly a formal model for describing families of models. Then we describe the first approach (cf. Ref. [7]) which is roughly based on imposing a membership mechanism to classify object collections as meta-model conformant or not. The second approach (cf. Ref. [2]), instead, tries to provide directly a signature whose terms are all and only meta-model conformant configurations by construction. After a discussion we propose a conciliation of both approaches fundamentally based on the first approach and inheriting the explicit treatment of structural containment from the second approach.

5.1 The need of meta-models

The theories of plain and nested object collections presented in the previous section can be considered as a sort of meta-model. However, consider the following object configuration:

```

< 'wshop3 : Room | state: ready , ends(rooms): 'item3 , 'store2 >
< 'item3 : Tool | alarm: off , opp(rooms) : 'wshop2 , ends(rooms): 'wshop3>

```

Many inconsistencies are evident at first sight: objects with attributes of other classes, associations with bad domain or co-domain, cyclic containment, association ends missing, etc. But such “evidence” does not scale up to large graphs, even for expert eyes. As in any other language, modelling languages need formal typing mechanisms to explicitly define well-formedness and tools to detect or prevent inconsistencies. This is the main role of meta-models. In our case, since a model is any term of the object-based signature, well-formedness is membership to a particular subsort \mathcal{S}

of `Model`. Next sections introduce two approaches to define and decide membership to `S`: the first one is based on membership predicates, while the second one consists on defining the constructors of `S`.

5.2 *Meta-Models as membership equational theories refining an object-based signature*

One of the most widely adopted modelling languages within the software industry is UML. Among the various UML dialects, class diagrams provide a suitable meta-modelling mechanism to define the structure of a software model at the very abstract level. A class diagram declares a basic alphabet of software components (classes), their internal properties (attributes) and their interrelations (associations). Such basic alphabet can be decorated with further information (e.g. the cardinality of relations, whether they are ordered or not, etc.) and equipped with further constraints expressed in languages such as the Object Constraint Language (OCL). A similar situation is present in many other modelling languages, e.g. in Architecture Description Languages (ADLs), which also provide some means of defining the basic architectural building blocks (component and connector types) and subtle architectural constraints. Even more, meta-modelling technologies such as the Meta-Object Facility provide a language for defining meta-models (including UML and MOF itself) based on the same idea: a diagram of attributed entity classes, their allowed relations and some additional constraints.

One of the basic ideas underlying the formalisation of the MOF within rewriting logic^[7] is the analogy between constructors/membership and alphabet/constraints. Indeed, the signature of object collections provides a generic language to describe any possible model. Introducing concrete class identifier constructors and attributes corresponds to depicting the diagram classes with their attributes and relations. Further constraints are specified by means of membership predicates on individual objects or a whole model. In our case a membership predicate is of the form $\text{cmb } M : S \text{ if } p$ where M is a `Model`-sorted term, S is the designated sort for meta-model conformance and p is the predicate that characterises conformance.

Consider, for instance the class diagram of Fig.4 which acts as informal meta-model of our running example. The formal meta-model introduces appropriate class identifier constructors (e.g. `Room`), datatype attribute names (e.g. `alarm`) and domains (e.g. sort `Alarm` with constructors `on`, `off`), and association attributes (e.g. `rooms`). The conformance predicate must check the correspondence of attributes and classes (e.g. `alarm` is a valid attribute for `Room` objects only), the domain of relations (e.g. `rooms` must refer to a set of objects of class `Room` only) and further relation constraints (e.g. `rooms` is a composition association as denoted by the diamond ended arrow).

The latter example of relation constraint regards one of the central issues of our work: how do we represent composition associations? Such relations represent structural containment and are hence required to be irreflexive up to transitive closure (to avoid self-containment) and injective (to ensure unique containers). In a plain graph representation checking those properties amounts to verify whether the composition graph (the graph obtained by dropping any edge not corresponding to a composition association) forms a forest (i.e. a set of trees).

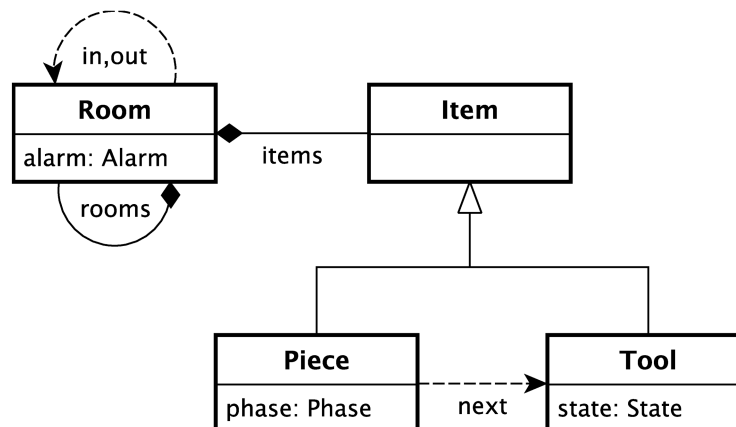


Figure 4. UML metamodel

5.3 Meta-Models as many-sorted algebras over an object-based signature

The approach we describe in this section stems from a tradition based on the use of grammars to recognise program shapes (e.g. architectural styles, workflow patterns). In particular, some authors promoted the use of graph grammars to define architectural styles and decide style conformance^[19]. With the observation that a context-free grammar can be turned into a many-sorted algebra, approaches were developed where a meta-model is roughly provided by an algebra that allows one to build conformant models only^[6].

In terms of our signature of object collections, the idea is to define the constructors for the sort S of conformant models. Obviously, this approach has some expressiveness limitations (it is not always possible or convenient to define a finite signature for a given sort) but has been proven very useful for typical model families such as client/server architectures, workflow patterns, and network topologies. Consider for instance, that our meta-model imposes the following constraints: rooms are flat (they do not contain other rooms), rooms are connected together with unidirectional transportation chains (cf. *in* and *out* relations in Fig.4), transportation chains connect all rooms in a sequential manner, no piece has a *next* attribute (as operations are dictated by the room sequence and the workpieces within each room).

Now a signature must be developed that contains one object constructor per class to ensure a correct use of attributes and a correct domain for references. Pieces, for example, can be constructed with operation `piece : Oid Phase -> PieceObj`, defined as follows:

```
eq piece(oid1,phase1) = < oid1 : Piece | phase: phase1 > .
```

This approach hampers the use of association pairs as this would require the related objects to be built together. Hence, relations are represented in one of the ends only (as references) and are typically free of constraints. Composition associations are dealt with nesting attributes as illustrated in Section 4. For instance, if we let `ItemsConf` denote the sorts for configurations made of items only, the constructor for room objects is `room : Oid Alarm Oid Oid ItemsConf -> RoomObj` defined as

```

eq room(oid1,alarm1,oid2,oid3,items1) =
  < oid1 : Room | alarm: alarm1 ,
    ends(in): oid2 , ends(out): oid3 , cont(items): items1 > .

```

A room configuration is a sequence of rooms built using individual rooms or a sequential composition operation $_ ; _ : \text{RoomConf } \text{RoomConf} \rightarrow \text{RoomConf}$ (cf. Fig.5):

```

eq conf1 < oid1 : Room | attSet1 , out: none > ;
  < oid2 : Room | attSet2 , in: none > conf2 =
  conf1 < oid1 : Room | attSet1 , out: oid2 >
  < oid2 : Room | attSet2 , in: oid1 > conf2 .

```

Conformant models are built with RoomConf-sorted configurations only:

```

op << _ >> : RoomConf -> S .

```

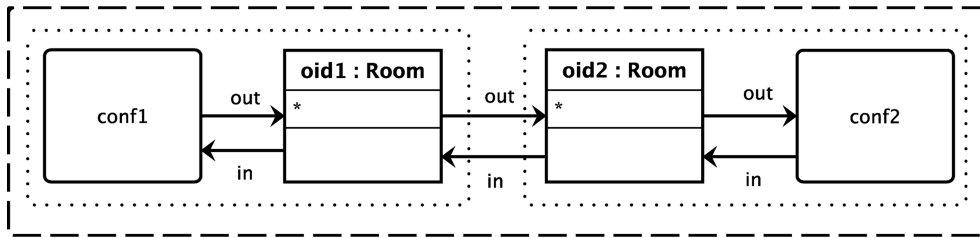


Figure 5. Sequential composition of rooms

5.4 Discussion: meta-model constructors or membership predicates?

Both approaches to meta-modelling share a common alphabet (the signature of object collections) and the idea of declaring a sort S of conformant models. The main difference is that the first approach decides membership to S by checking a set of predicates, while the second approach defines the constructors for S .

The main advantages of the first approach are that it more suitably applies to standard modelling technology like UML and, in particular, MOF. Indeed, even if not discussed here the approach is reflective: there is a boxes-and-lines diagram (the meta-meta-model called M3 in MOF) that can describe itself and is used to describe meta-models (such as UML). This tower of meta-modelling levels is captured by the reflection mechanism of rewriting logic^[7] (efficiently implemented in Maude). The second approach, in place, has a unique advantage: models are conformant by construction, which not only spares the effort of checking meta-model constraints but can facilitate the definition of conformance-perserving model transformations at high-level of abstraction.

The signature of nested object collections exploits best features of both approaches, by capturing the constraints inherent to composite aggregations with nesting constructors. In that way, one obtains several benefits (no need to check containment constraints) without a price to pay since, as shown in the previous section, plain and nested representations are in bijective correspondence.

6 Programming Model Transformations

The need for visual modelling languages and the graph-based nature of models have contributed to the success of graph transformation approaches to model-based transformations. In such approaches, transformations are programmed in a declarative way by means of a set of (graph) rewrite rules. This section discusses two rule-based approaches to the definition of model transformations. Both consist on defining a rewrite theory over (possibly nested) object collections. The main difference between the approaches is the format of rules. The first approach is based on the tradition of single-pushout graph transformation^[22], while the second one stems from the tradition of *Structural Operational Semantics* (SOS)^[21].

6.1 Graph transformation style

The transformation style that we consider here is based on the single-pushout graph transformation approach. The main idea is that each rule has a left-hand side and a right-hand side pattern. Each pattern is composed by a set of objects (nodes) possibly interrelated by means of association ends (edges). A rule can be applied to a model whenever the left-hand side can be matched with part of the model, i.e. each object in the left-hand side is (injectively) identified with an object and idem for the association ends. The application of a rule removes the matched part of the model that does not have a counterpart in the right-hand side and, vice versa, adds to the model a fresh copy of the right-hand side part that is not present in the left-hand side. Items in common between the left-hand side and the right-hand side are preserved during the application of the rule. Very often, rules are equipped with additional application conditions, including those typical of graph transformation systems (e.g. no dangling edges) and its extensions like *Negative Application Conditions* (NACs).

In our setting, this means that rules have in general the following format:

```

r1 << lhs conf1 >> => << rhs conf1 >>
  if applicable(lhs conf1) .

```

where `lhs` and `rhs` stand for the rule's left- and right-hand side object configurations, `conf1` acts as the context in which the rule will be applied (i.e. the rest of the object configuration), and `applicable` is the boolean function implementing the application condition. Simpler forms are possible, e.g. in absence of application conditions the context is not necessary and rules take the form: `r1 lhs => rhs` .

To illustrate the general rule format, we consider first the plain representation of models based on object collections. A simple reconfiguration rule is illustrated in Fig.6. The rule models the transportation of a workpiece `p1` from its current location `r1` to the location `r2` of the tool `t2` that must operate on it (as indicated with the `next` relation). The rule abstracts away from the actual room-by-room physical transportation and just replaces the `items` edge from `r1` to `p1` by a new such edge from `r2` to `p1`:

```

r1 << < r1 : Room | alarm: off , ends(items): p1 items1 , attSet1 >
  < p1 : Piece | opp(rooms): r1 , ends(next): t2 , attSet2 >
  < r2 : Room | alarm: off , ends(items): t2 items2 , attSet3 >
  < t2 : Tool | state: ready , opp(items): r2 , attSet4 >
  conf1 >> =>
  << < r1 : Room | alarm: off , ends(items): items1 , attSet1 >

```

```

< p1 : Piece | opp(rooms): r2 , ends(next): t2 , attSet2 >
< r2 : Room | alarm: off , ends(items): t2 p1 items2 , attSet3 >
< t2 : Tool | state: ready , opp(items): r2 , attSet4 >
conf1 >> .

```

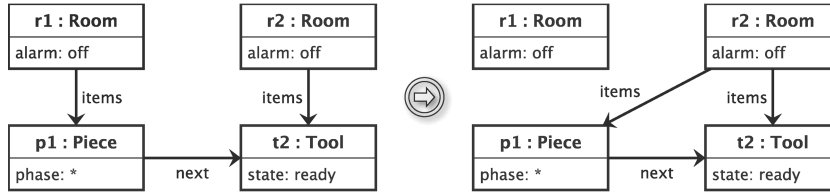


Figure 6. Reconfiguration rule transporting a workpiece to the location of its next tool

Let us now consider the hierarchical representation of models based on nested collections of objects. The reconfiguration example we consider is a migration rule that models the evacuation of all items within a compromised room (`alarm: on`) to a safe one (`alarm: off`). Figure 7 illustrates the rule:

```

r1 << < r1 : Room | alarm: on , cont(rooms): rooms1 , cont(items): items1 >
< r2 : Room | alarm: off , cont(rooms): rooms2 , cont(items): items2 >
conf1 >> =>
<< < r1 : Room | alarm: on , cont(rooms): rooms1 , cont(items): none >
< r2 : Room | alarm: off , cont(rooms): rooms2 , cont(items): items1 items2 >
conf1 >> .

```

It is worth to observe, that these kind of rules have some limitations that are particularly relevant in the hierarchical representation: the nesting depth is fixed. In the above example, for instance, it applies to top-level rooms only. This problem is not present in the flat representation, though both suffer from another relevant issue: the coordination of transformations involving an arbitrary number of objects. The rule format we shall see in the next section overcomes such drawbacks.

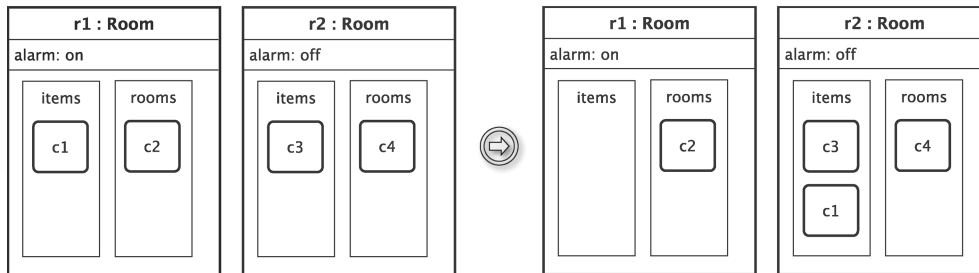


Figure 7. Reconfiguration rule evacuating items from a compromised room to a safe one

6.2 Structural operational semantics style

In this section we describe transformation rules in the style of *Structural Operational Semantics*^[21] (SOS). The basic idea is to define a model transformation by structural induction, which in our setting basically amounts to exploiting set union and (possibly) nesting.

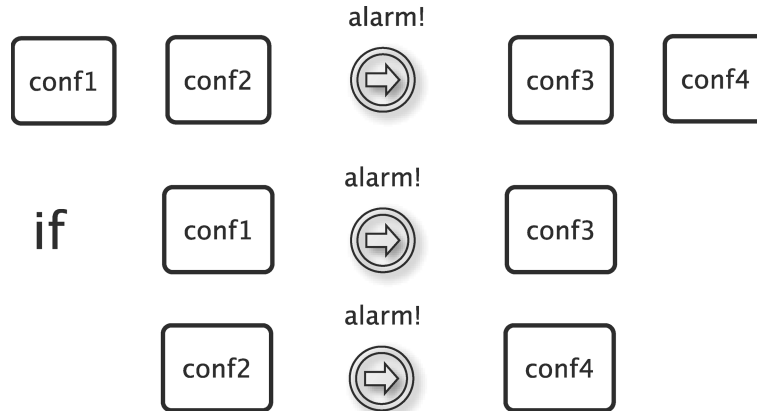


Figure 8. Reconfiguration rule propagating the alarm within multiset union

Before introducing the different rule formats and illustrating an example we recall that SOS rules make use of rule labels to coordinate rule applications. We follow the implementation style of SOS semantics in rewriting logic described in Ref. [10] by enriching our signatures with sorts for rule labels (Lab), label-prefixed configurations LabConf , and a constructor $\{-\}_- : \text{Lab Conf} \Rightarrow \text{LabConf}$ for label-prefixed configurations. In addition, we enforce rule application at the top-level of terms only (via Maude's `frozen` attribute) so that sub-terms are rewritten only when required in the premise of a rule (as required by the semantics of SOS rules). With this notation a term $\{\text{lab1}\}\text{conf1}$ represents that a configuration `conf1` obtained after application of a `lab1`-labelled rule.

We start by considering a flat representation of models, where the only structure is that of set union. The most typical rule format allows us to conclude a transformation `lab1` for a configuration made of two parts `conf1` and `conf2` provided that each part can respectively perform some transformation `lab2`, `lab3`:

```
cr1 conf1 conf2 => {lab1} conf3 conf4
if conf1 => {lab2} conf3
/\ conf2 => {lab3} conf4 .
```

As an example consider a reconfiguration that triggers the alarm of all rooms. Various rules are needed. First, we need rules for individual room objects to declare their ability to turn their alarm on, and items to agree on that. For simplicity we consider that any object is ready to accept the transformation:

```
r1 obj1 => {alarm!} obj1 .
```

The interesting rule is illustrated in Fig.9 which allows us to exploit union set:

```
cr1 conf1 conf2 => {alarm!} conf3 conf4
if conf1 => {alarm!} conf3 .
/\ conf2 => {alarm!} conf4 .
```

Consider now a hierarchical representation of models based on nested object collections. In this situation we need rules for dealing with nesting. Typically, the needed rule format is the one that defines the transformation `lab1` of an object `oid1` conditioned to some transformation `lab2` of one of its contents `contid1`:

```

cr1 < oid1 : cid1 | cont(contid1): conf1 , attSet1 > =>
  {lab1} < oid1 : cid1 | cont(contid1): conf2 , f(attSet1) >
  if conf1 => {lab2} conf2 .

```

Such rules might affect the attributes of the container object (denoted with function f) but will typically not change the object's identifier or class (unless we are dealing with a model refactoring or translation). More elaborated versions of the above rule are also possible, for instance involving more than one object or not requiring any rewrite of contained objects.

In our example of the alarm propagation, the rule we need is illustrated in Fig.9 and textually defined as:

```

cr1 < r1 : Room | alarm: alarm1 , cont(rooms): rooms1 ,
cont(items): items1 > =>
  {alarm!} < r1 : Room | alarm: on , cont(rooms): rooms2 , cont(items): items1 >
  if rooms1 => {alarm!} rooms2 .

```

Finally, rules are need to close the transformations at the level of models. Such rules have the following format:

```

cr1 << conf1 >> => << conf2 >>
  if conf1 => {lab1} conf2 .

```

In our example the rule for alarm propagation would be

```

cr1 << conf1 >> => << conf2 >>
  if conf1 => {alarm!} conf2 .

```

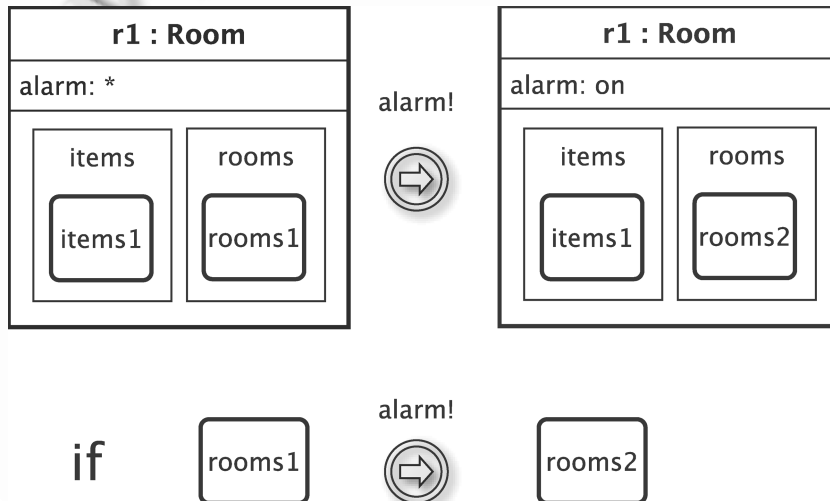


Figure 9. Reconfiguration rule propagating the alarm within the nesting structure

6.3 Discussion: Graph transformation or SOS style?

The presented rule formats are well-known in Computer Science. For instance, in the field of programming languages semantics they essentially correspond to reduction and transition label semantics of process algebras. As in almost any choice of techniques, there is a killer application for any style. For instance, a plain graph representation and a graph-like rule format is best suited for those transformations

involving a fixed number of inter-related, arbitrarily composed objects. This is for instance evident in the example of single workpiece transportation (cf. Fig.6). Just one rule is needed with the mentioned representation and rule format. Consider, instead, the hierarchical representation based on nested object collections. The number of possible situations is infinite: the room of the workpiece can contain the room of the machine tool at any depth, or vice versa, or they could not contain each other but be contained at different depths of the production plant. Therefore it is not possible to identify a context for a reduction rule. SOS rules are needed in this case, in the very same way as action synchronisation in process algebras with nesting features (e.g. ambients or sessions).

On the other hand, a hierarchical representation and a SOS-like rule format is best suited for those transformations guided by the nesting structure and involving an arbitrary number of objects. For instance, if we want to define a reconfiguration turning all alarms on or off in one shot (cf. Fig.9). Instead, the graph-like transformation requires a cumbersome set of rules to deal with the possibility that some rooms are not ready to perform the required reconfiguration. Indeed, some similar reconfigurations (like reversing a chain) are known not to be programmable with this rule format.

Apart from the above mentioned differences, the graph-like approach has as main advantages its proliferation among the community of graph transformation and visual modelling languages and the fact that transformations are sequential: rule applications are composed one after the other. On the other hand the SOS style offers a familiar approach to members of the formal methods and programming language semantics communities and provides tree-shaped transformations (since rule derivations are inductively defined) that facilitate their layered understanding.

Of course many other formats have been proposed in the literature and among them it is worth mentioning Synchronised Hyperedge Replacement^[13] (SHR), which mixes SOS ingredients and graph productions. SHR is a graph-based framework for modelling the operational semantics of systems with mobility and multiple synchronisation. Several flavours of SHR semantics exist, but have as a common core the idea of defining the transformation of a large graph by combining local transformations that apply to single edges. The nodes of the graph are responsible for coordinating the interaction of all the edges that are attached to them. However, the basic structure taken into account by SHR rules is that of graph union and locality of nodes. In this sense, our approach can be seen as extending SHR to deal with nesting and more general structures.

7 Executing Model Transformations

This section presents some experimental results intended to raise the interest on performance issues when executing model transformations, depending on the chosen representation and rule format. As a test case we use the evacuation scenario of our running example, where items must be transported from compromised rooms (those with alarm *on*) to safe ones (those with alarm *off*). We consider an additional **status** attribute for items (that was not introduced in previous sections for the sake of simplicity). It can take various values, including **safe** (whenever they the item is safe).

The approach based on a flat representation is given by a set of SPO-like rules. The most significant rule is similar to the ones discussed in Section 3 and is depicted in Fig.10. The rule considers a compromised room $r1$ and a safe room $r2$ that are neighbours (they have a common container room $r0$). The rule moves one item $i1$ from the compromised room to the safe one, while setting its status attribute to **safe**. Some more rules are needed (for instance for considering top-level rooms without containers) and some of them have application conditions. As a consequence, the applicability of those rule requires to check the whole model and we have no guidance on which rules to apply first. The safe system (the system without items in need of evacuation) is reached when no more transformation rules are applicable.

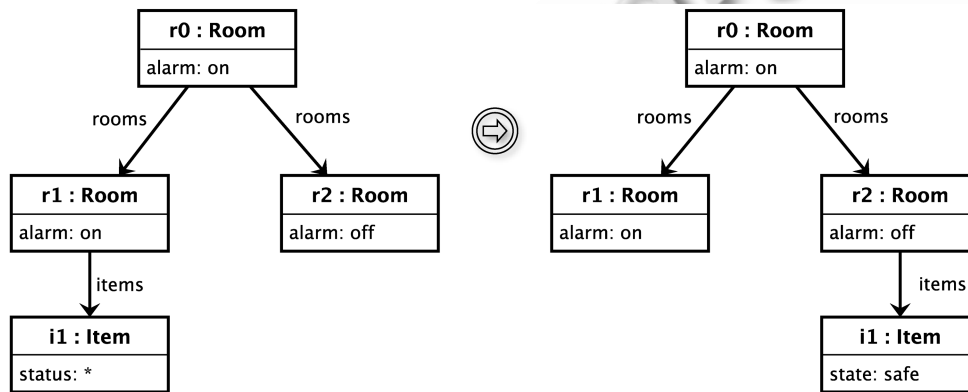


Figure 10. SPO reconfiguration rule evacuating an item from a compromised room

The structured transformation is based on a hierarchical representation and conditional rules. Figure 11 illustrates the main rule: all the items $c1$ of a compromised room $r1$ are evacuated into a safe neighbor room $r2$, while setting their status to **safe** inductively (via **safe!**-labelled rules).

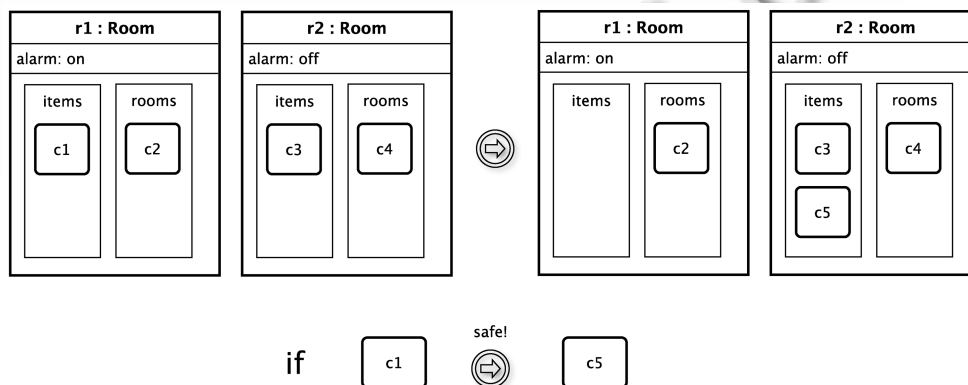


Figure 11. SOS reconfiguration rules evacuating all items from a compromised room

We are not going into further details of each transformation, but we offer some simple experimental results to provide evidence of the advantages of our proposal in some cases.

In order to test the scalability of both transformation approaches we have prepared an automatic generator of production plants. The generator is parametrizable to allow us to generate different test cases. Here we consider a simple case with a single parameter which is the depth of the room containment tree, i.e. for a given natural number n , we generate a production plant organised as a binary tree of depth n . In those trees, leaves are items and the fathers are either compromised or safe rooms. The rest of the rooms are safe and the grandfather of leaves have exactly one safe room and one compromised room.

Experiments were run under Mac OS X on a MacBook with a 2.0Ghz dual-core processor and 2GB of RAM. Each experiment consists on the transformation of an instance of our test case using the plain (SPO) and the structured transformation (SOS) described above. For each experiment we have recorded the number of rewrites and the running time. Space consumption was not measured as Maude does not provide such information. Each experiment is performed for increased size factor n , starting with $n = 2$ and ending with $n = 8$ (recall that each model has size of order 2^n). The goal of the experiment is to collect evidence of the fact that structure-driven models and transformation rules can lead to a considerably more efficient model transformation.

The results of Fig.12 show a clear superiority of structured model transformation both in terms of running time and number of rewrites. These results confirm our expectation. Indeed, in this particular test case the situation can be roughly explained as follows. In most cases matching a rule consists on finding a subtree whose root is a room having two subtrees: one having a compromised room as root and one having a safe room as root. In the SPO case the tree is not parsed: indeed we are given a graph and have to check all possible subset of nodes to see if they constitute indeed a tree. Instead in the SOS case the tree is already parsed (the parsing is term of the hierarchical representation) which enormously facilitates rule matching. As a consequence, the SPO transformation involves more unsuccessful rule attempts and this is the main reason of the drastic difference in running time (and not in number of effective rewrites).

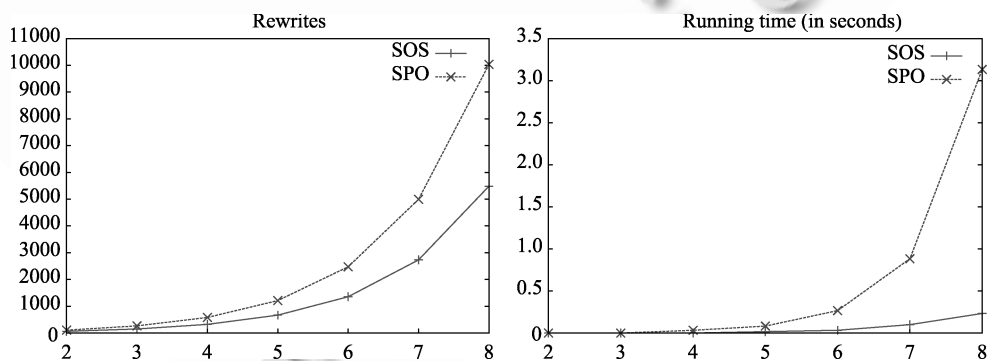


Figure 12. Performance evaluation of flat (SPO) and structured (SOS) model transformations

This small experiment is not intended to claim that structured transformations are more performant in general, but to provide evidence that performance is a key issue when executing transformations or analysing them. We believe that these issues deserve further consideration and investigation and the possibility of moving from

one representation style to the other in a seamless way can be profitably exploited to select each time the more convenient approach.

8 Conclusion

We have presented an approach for the description and analysis of model- and rule-based specifications with hierarchical structure. Our approach provides several benefits. First, it is built over the solid foundation of algebraic approaches like rewriting logic, structural operational semantics. Second, all the mathematical machinery is presented in the unifying, tool-supported framework of rewriting logic. Third, the approach fits perfectly with MOF-based technology as the MOF structure is somewhat homomorphic with our formalism. As a matter of fact our approach can be understood as a no-harm enhancement of the algebraic approach to MOF of Ref. [7]: one can pass from a *composition-as-relation* representation to a *composition-as-containment* representation in a bijective manner, to use structural induction there where convenient. Fourth, the approach imposes a design discipline based on the hierarchical structure of composition associations, which contributes to the scalability of model- and rule-based approaches. Indeed, designers can benefit from the layered view introduced by the hierarchical structure and structured rewrite rules can lead to more efficient analysis activities.

Acknowledgments

The authors are grateful to Artur Boronat for fruitful discussions that have inspired this paper.

References

- [1] Boronat A, Bruni R, Lluch-Lafuente A, Montanari U, Paolillo G. Exploiting the hierarchical structure of rule-based specifications for decision planning. In: Hatcliff J, Zucca E, eds. Proc. of the International Joint Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE'10). LNCS 6117, 2009. 455–469.
- [2] Bruni R, Hölzl MM, Koch N, Lluch-Lafuente A, Mayer P, Montanari U, Schroeder Andreas, Wirsing M. A service-oriented UML profile with formal support. In: Baresi L, Chi CH, Suzuki J, eds. Proc. of the 7th International Joint Conference on Service-Oriented Computing (ICSOC-ServiceWave'09). LNCS 5900, 2009. 455–469.
- [3] Boronat A, Heckel Reiko, Meseguer J. Rewriting logic semantics and verification of model transformations. Proc. of the International Conference on Fundamental Aspects of Software Engineering (FASE'09). LNCS 5503, Springer, 2009.
- [4] Bruni R, Lluch Lafuente A. Ten virtues of structured graphs. Proc. of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques 2009 (GT-VMT'09). Volume 18 of Electronic Communications of the European Association of Software Science and Technology, ECEASST, 2009.
- [5] Bruni R, Lluch Lafuente A, Montanari U. Hierarchical design rewriting with Maude. Proc. of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08). volume 238(3) of ENTCS, Elsevier, 2008. 45–62.
- [6] Bruni R, Lluch Lafuente A, Montanari U, Tuosto E. Style based architectural reconfigurations. Bulletin of the European Association of Theoretical Computer Science (EATCS), 2008, 94: 161–180.
- [7] Boronat A, Meseguer J. An algebraic semantics for MOF. Proc. of the International Conference on Fundamental Aspects of Software Engineering (FASE'08). LNCS 4961, Springer, 2008. 377–391.

- [8] Broy M. Modular hierarchies of models for embedded systems. *ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'03)*, 2003, 0: 183.
- [9] Broy M. Two sides of structuring multi-functional software systems: function hierarchy and component architecture. *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*. IEEE Computer Society, 2007. 3–12.
- [10] Broy M. Seamless model driven systems engineering based on formal models. *11th International Conference on Formal Engineering Methods (ICFEM 2009)*. LNCS 5885, Springer, 2009. 1–19.
- [11] Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott CL. All about maude. LNCS 4350, Springer, 2007.
- [12] Ehrig H, Ehrig K, Prange U, Taentzer G. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006.
- [13] Ferrari GL, Hirsch D, Lanese I, Montanari U, Tuosto E. Synchronised hyperedge replacement as a model for service oriented computing. *Proc. of the 4th International Symposium Formal Methods for Components and Objects (FMCO'05)*. LNCS 4111, Springer, 2005. 22–43.
- [14] Fradet P, Le Métayer D. Shape types. *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. New York, NY, USA, ACM Press, 1997. 27–39.
- [15] Hirsch D, Inverardi P, Montanari U. Graph grammars and constraint solving for software architecture styles. *Proc. of the International Software Architecture Workshop (ISAW'98)*. ACM Press, 1998. 69–72.
- [16] Hirsch D, Montanari U. Shaped hierarchical architectural design. *ENTCS*, 2004, 109: 97–109.
- [17] Krüger I, Prenninger W, Sandner R, Broy M. From scenarios to hierarchical broadcasting software architectures using UML-RT. *International Journal of Software Engineering and Knowledge Engineering*, 2002, 122: 155–174.
- [18] Krüger I, Prenninger W, Sandner R, Broy M. Development of hierarchical broadcasting software architectures using UML 2.0. In: Ehrig H, Damm W, Desel J, Große-Rhode M, Reif W, Schnieder E, Westkämper E, eds. *SoftSpez Final Report*. LNCS 3147, Springer, 2004. 29–47.
- [19] Le Métayer D. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 1998, 24(7): 521–533.
- [20] Meseguer J. Conditional rewriting logic as a united model of concurrency. *Theoretical Computer Science*. 1992, 96(1): 73–155.
- [21] Plotkin GD. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 2004, 60-61: 17–139.
- [22] Rozenberg G. *Handbook of Graph Grammars*. World Scientific, 1997.
- [23] Verdejo A, Martí-Oliet N. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 2006, 67(1-2): 226–293.