Precise Graphical Representation of Roles in Requirements Engineering

Pavel Balabko, Alain Wegmann
Laboratory of Systemic Modeling, Ecole Polytechnique Fédérale de Lausanne, EPFL-IC-LAMS, Lausanne, Switzerland

Email: pavel.balabko@epfl.ch, alain.wegmann@epfl.ch

Abstract: Modeling complex systems can not be done without considering a system from multiple views. Using multiple views improves model understandability. However the analysis of models that integrate multiple views is difficult. In many cases a model can be evaluated only after its implementation. In our work we describe a visual modeling framework that allows for the evaluation of multiview models. We give an overview of our framework using a small case study of a Simple Music Management System. For each view in our framework we specify a separate role of the system. The whole system is specified as a composition of smaller roles. Each role, as well as the whole model of a system, can be evaluated by means of the analysis of possible instance diagrams (examples) that can be generated automatically. Instance diagrams are generated based on the formalization of visual models using the Alloy modeling language.

1 Introduction

Modeling of complex systems from multiple views is unavoidable. Multiple views help in solving the scalability problem described in [Chang99] and improve model understandability: each view can be analyzed independently of other views. However, the reasoning about the whole model as a composition of multiple views is difficult. It is difficult because the same entity in the Universe of Discourse (UoD or reality) can be modeled as several independent model elements. The role of modeler in this case is to make one model from the set of models corresponding to different views. A modeler has to find identical model elements (elements that model the same entity in the UoD) in models to be composed and make sure that a resulting model makes sense (it reflects "reality" or the UoD in some meaningful way). Here we will not talk about how a modeler should search for the identity of model elements. This is a separate research topic that is based on the Tarski declarative semantics: the semantics that defines equivalence of an agreed conceptualization of the UoD to a concrete concept in the model (see [Naumenko02] for details). We concentrate on the second part of the problem: assuming that we agree what the identity of model elements means, we investigate how to ensure that a composed model gives us some adequate reflection of reality.

The design of software systems should be based on adequate models (models that help to solve someone problems). Only adequate models can ensure that a system evolves in correspondence with the system's users needs. Therefore early system requirement specification is an important step in the evolution of software systems. In our work we concentrate on the very early model validation of system requirements that results in an adequate model. The understanding of model adequacy requires the interaction between system developers and customers (or other system stakeholders). They are not experts in system modeling and may have problems in understanding semantics of specification diagrams (like UML class or statechart diagrams). These diagrams are convenient for professionals and represent the generalizations of many examples (or scenarios) from the problem domain. On the contrary, for customers it is more convenient to talk in terms of particular examples (or instances) of models. In our work we describe a visual modeling framework that allows for the composition of models from different views and the analysis of composed models based on automatically generated model instances.

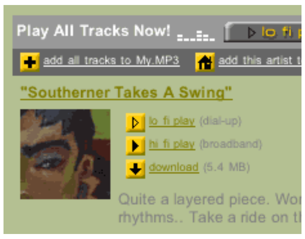
Our framework supports modeling with multiple views. For each view we specify a separate role for a system. The whole system is specified as a composition of smaller roles. Each role as well as the whole model of a system can be evaluated by means of the analysis of the possible instance diagrams (examples) that can be generated automatically. Instance diagrams are generated using the Alloy

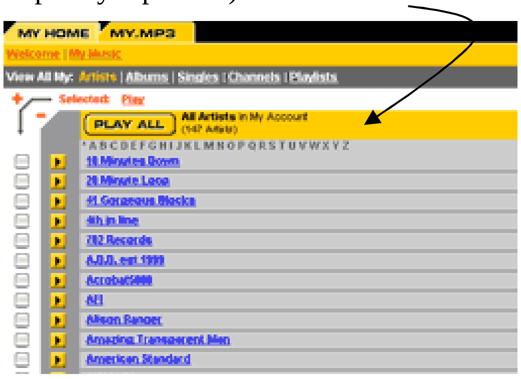
Constraint Analyzer[1] [Jackson02]. It is based on the Alloy modeling language. It is a simple structural modeling language based on first-order logic. Alloy models are analyzed using the Alloy Constraint Analyzer. The analyzer attempts to find a solution based on a given specification. If a solution is found, it displays it as a graph of nodes and edges (similar to an instance diagram).

We describe our framework using a small case study of the Simple Music Management System (a simplified version of www.mp3.com). We took mp3.com as a base of a Simple Music Management System because its model is a complex one and can not be modeled from a single point of view. We consider two major views for this system: User View and Artist View. Here we give a brief description of these two views:

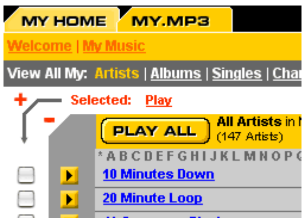
MP3.com users can:

- Find free artist albums and artist singles in places like MP3.com artist pages (pages containing artist albums, singles etc), and add them to My Music (personal user music collection at http://my.mp3.com).





- Play music from My Music.



- Manage personal user music collection (My Music):
  - Manage user play lists. A user play list can include single tracks (a user track that is not part of any other user album) or album tracks from the personal user music collection (My Music).
  - Delete user playlists, singles and albums from My Music.

MP3.com artists can:

- With a Standard Content Agreement:
  - Create one or more Artist Pages where an artist can post his materials
  - Artist materials can include: artist singles and artist albums with album tracks.

The structure of this paper is the following. In section 2 we give the minimum set of concepts necessary for the understanding of our paper. In section 3 we show how a model of a Simple Management Music System can be built in our framework: in section 3.1 we begin with the specification of base roles, in section 3.2 we specify a composition of base roles and in section 3.3 we show how a composed model can be evaluated through the analysis of automatically generated instance diagrams. Section 4 is a conclusion.

2 Definition of Main Concepts

In this section we present concepts that we use in our work. In order to give rigorous definitions for these concepts, we have to choose a consistent semantic framework. We use the ISO/ITU standard

[1] See http://sdg.lcs.mit.edu/alloy/

“Reference Model for Open Distributed Processing” – part 2 [ISO96] as a basis of our modeling framework.

Based on RM-ODP, modeling consists of identifying entities in the universe of discourse and representing them in a model. The *universe of discourse* (UoD) corresponds to what is perceived as being reality by a developer or a customer; an *entity* is “any concrete or abstract thing of interest” [ISO96] in the UoD. Identified entities are modeled as *model elements* in a *model*. Model elements are different modeling concepts (object, action, behavior etc). We give definitions of some modeling concepts necessary for the understanding of our paper (other definitions see in the RM-ODP). We begin with the definition of an object. If in the UoD we have entities that can be modeled with state and behavior, we model these entities as objects:

Object: *“An object is characterized by its behavior and dually by its state”* [ISO96].

The duality of state and behavior means that the state of an object determines the subsequent behavior of this object. To specify state and behavior of an object we will use Object Behavior and Object State Structure:

Object Behavior Structure: *A collection of actions and a set of (sequential) relations between actions.*

All possible state of a system we model as a state structure:

Object State Structure: *A collection of attributes, attribute values and relations between attributes.*

Attributes can change their values; relations between attributes can be created or deleted.

Based on the definition of behavior we define a role:

Role: *“An abstraction of the behavior of an object” intended for achieving a certain common goal in collaboration with other roles.*

Like a behavior of an object is defined dually with a state of an object, a role is also defined with its state. To specify possible states of a role we use Role State Structure:

“Role State Structure *is a subset of the complete state of a system that is used for reasoning about a given role”.*

In this work we consider modeling and analysis of a role state structure. Graphically we represent a role state structure in the following way: each role we represent as a box with two panes (see figure 1). The upper pane indicates the name of a role. The lower pane contains a role state structure. To represent it we use graphical notation inspired by a UML class diagram.

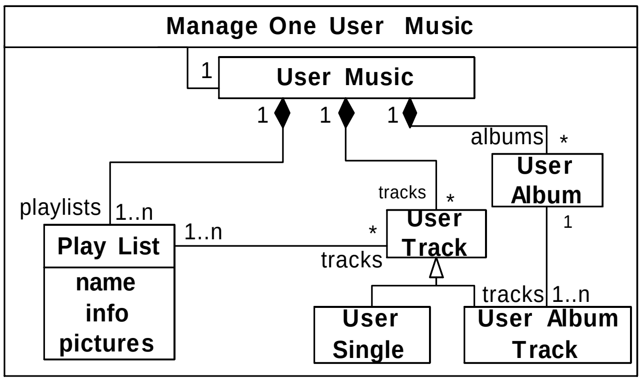
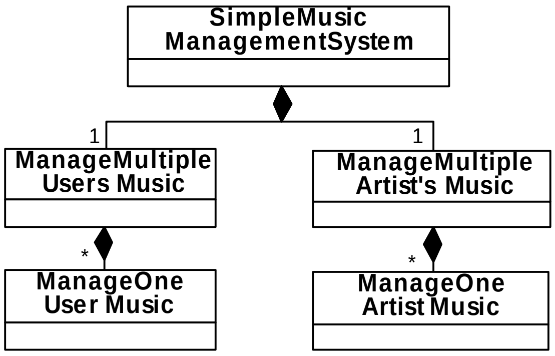


Figure 1. Graphical notation that shows a role state structure for the “Manage One User Music” role.

The main difference with a UML class diagram is that we make explicit belonging of attributes to roles. We do it by means of the notation inspired by the Catalysis method [D'Souza98]. We connect attributes, that are not parts of any other attribute, to the border of a lower pane box. In figure 1, for example, we explicitly connect the “User Music” attribute to the “Manage One User Music” role. This allows us to specify that this role always has one “User Music” attribute.

3 Model of the Simple Music Management System and its Analysis

This section is the main part of our paper where we describe our modeling framework based on the example of a model of a Simple Music Management System. In section 3.1 we begin modeling of a Simple Music Management System with two base roles: “Manage One User Music” and “Manage One Artist Music” (roles in the lower part of the role hierarchy in figure 2). The first role is a role of the system from the point of view of one user that manages his music (My Music). The second role is a role of the system from the point of view of one artist that manages his music to be provided for users.



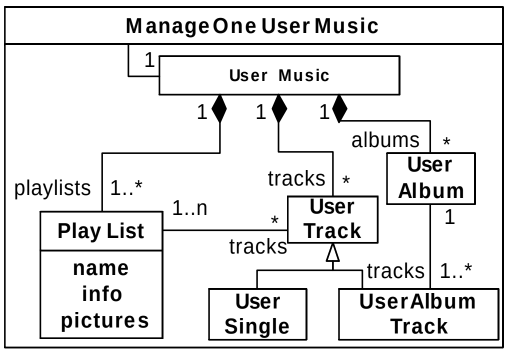
SimpleMusic ManagementSystem
1
1
ManageMultiple Users Music
ManageMultiple Artist's Music
*
*
ManageOne User Music
ManageOne Artist Music

Figure 2 Hierarchy of roles for a model of a Simple Music Management System.

In section 3.1 we give models of two higher level roles in the role hierarchy: “Manage Multiple User Music” and “Manage Multiple Artist Music”. Both these roles are correspondingly composed from multiple lower-level roles: “Manage One User Music” and “Manage One Artist Music”. The first one specifies the system from the point of view where there are multiple users managing their music; the second role specifies the system from the point of view where there are multiple artists, managing their music. In section 3.2 we explain how a composition of the “Manage Multiple User Music” and “Manage Multiple Artist Music” roles can be specified. In section 3.3 we explain how a composed model can be evaluated based on the analysis of automatically generated instance diagrams.

3.1 Specification of Base Roles

We begin with the modeling of the state structure of the “Manage One User Music” role. For this role we give an Alloy code and explain how it corresponds to a model of this role (see table 1).



```
module Main/UserMusic/OneUserMusic   //module declaration
sig UserTrack {}
sig UserMusic { tracks: set UserTrack,
                playlists: set PlayList,
                albums: set UserAlbum}
sig UserAlbum {tracks: set UserAlbumTrack}
part sig UserSingle, UserAlbumTrack extends UserTrack {}
sig PlayList {tracks: set UserTrack}

// ------------------ Multiplicity Facts -------------------------
fact {   // PlayList (1..*) -> (*) UserTrack
    all ut:UserTrack | some pl:PlayList | ut in pl.tracks }

fact {   // UserMusic (1) -> (1..n) PlayList
    (all um:UserMusic | some pl:PlayList | pl in um.playlists) &&
    (all pl:PlayList | one um:UserMusic | pl in um.playlists) }

fact {  // UserMusic (1) -> (*) UserTrack
    all ut:UserTrack | one um:UserMusic | ut in um.tracks }

fact {  // UserMusic (1) -> (*) UserAlbum
    all ua:UserAlbum | one um:UserMusic | ua in um.albums }

fact { // UserAlbum (1) -> (1..*) UserAlbumTrack
(all ua:UserAlbum|some uat:UserAlbumTrack|uat in ua.tracks) &&
(all uat:UserAlbumTrack | one ua:UserAlbum | uat in ua.tracks) }
// ------------------ End of Multiplicity Facts -------------------------
```

Table 1 State structure model for the “Manage One User Music” role and the Alloy code.

Let’s look at the Alloy code in table 1. All rectangles (attributes) in the diagram from this table have corresponding type declarations in the code (lines 2-6). A user track can be either a user single (a track that is not a part of any user’s album) or a user album track. In code this is expressed as the partitioning of all user tracks into two sets (line 5).

A type declaration may introduce relations. For example the “User Music” type (line 3) introduces three relations (tracks, playlists and albums). The keyword *set* indicates multiplicity: it tells that a relation points to a set of elements. For example, for each element of the “User Music” type there is a set of elements of the “User Track” type. To make multiplicity more strict (like “1..*” instead of “*”), multiplicity facts are used (lines 7 - 18). For example, the fact in line 16 contains two constraints (line 17 and 18). The constraint from line 17 tells that for all elements of the “User Album” type there are some (at least one) elements of the “UserAlbumTrack” type in the “tracks” relation.

As we mentioned in the introduction, the simplest way to make an agreement with no professionals (like customers) that a model adequately represents the UoD is to consider model instances. Model instances can be generated using the Alloy Constraint Analyzer. It checks the consistency of a formal Alloy model; randomly generates a solution (an instance of a model) and visualizes it. In order to generate a solution, a scope of a model should be defined. A scope defines how many instances of each type a solution may include.

Examples from figure 3 and 4 were built with the Alloy Constraint Analyzer.

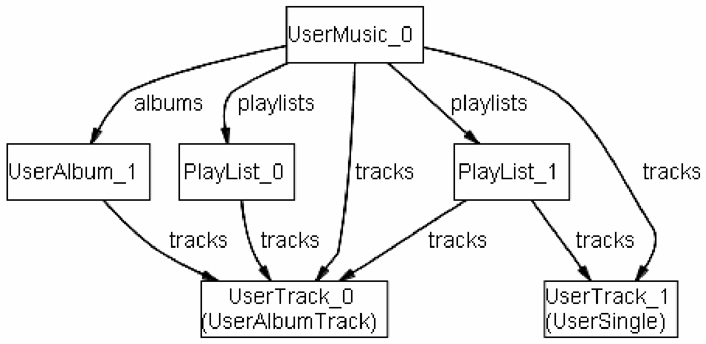


Figure 3 Example with a scope 2 but 1 UserMusic

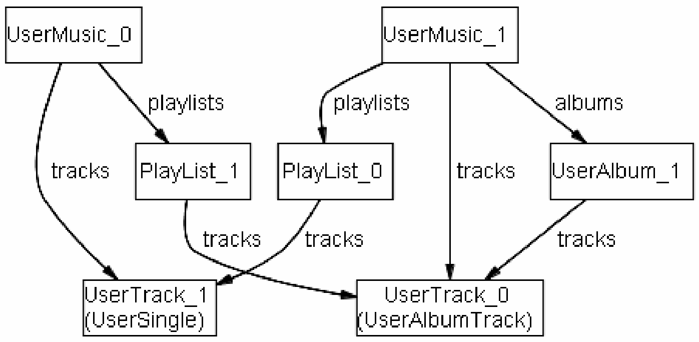
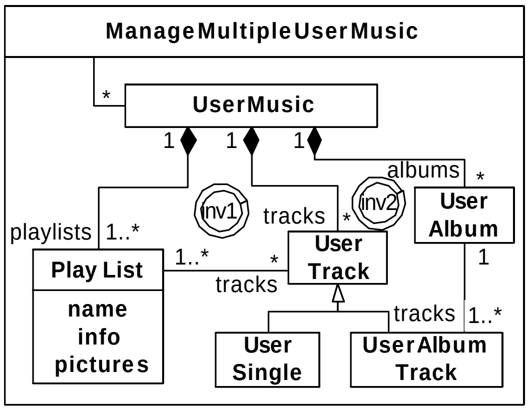


Figure 4 Example with a scope 2

The scope of the example in figure 3 was defined in the following way: 2 All, 1 UserMusic. This means that this example, generated by the Alloy Constraint Analyzer, could have maximum two elements of each type (except the UserMusic type) and one element of the UserMusic type. Having only one element of the UserMusic type corresponds to the fact that the “Manage One User Music” role includes one UserMusic attribute (see diagram in Table 1). The Alloy Constraint Analyzer allows for the generation of several different examples for the same model and the same scope. All these examples can be used to reach an agreement with a customer if the state structure model of the “Manage One User Music” corresponds to the common understanding of the UoD.

The next step will be to make a state structure model for the “Manage Multiple User Music” role (see figure in table 2). Comparing with the diagram from table 1, we changed the multiplicity of the association that connects the “Manage Multiple User Music” role with the UserMusic attribute (see figure in table 2). This allows for specifying that this role has multiple “User Music” attributes. All other elements of the diagrams from tables 1 and 2 are the same (for the moment we ignore *inv1* and *inv2* in the diagram from Table 2). This allows us to reuse the Alloy code from table 1 and test it for a new scope. With a new scope we have to require that a solution includes several (at least two) instances of the UserMusic type. Therefore we choose a new scope equal to two. This means a solution may include maximum two instances of each type.

A solution, found by the Alloy Constraint Analyzer for the Alloy code from table 2 with the scope equal to two, is shown in figure 4. We can see that this solution does not adequately show the reality: “A user play list may include single tracks or album tracks from this user music” (see the description of a Simple Music Management System in introduction). However, the solution, found by the Alloy Constraint Analyzer, shows that a user play list may include tracks of another user. Therefore we have to put additional constraints to make a state structure model for the “Manage Multiple User Music” correct. These additional constraints we put in a separate Alloy module: MultipleUserMusic (see Table 2).



```
module Main/UserMusic/MultipleUserMusic    //module declaration
open Main/UserMusic/OneUserMusic

// ----------------- Multiplicity Invariants ------------------------

fact inv1 {
   all um:UserMusic | all ut:UserTrack | (ut in um.playlists.tracks)
                                                  => (ut in um.tracks) }

fact inv2 {
   all um:UserMusic | all uat:UserAlbumTrack |
                      (uat in um.tracks)  => (uat in um.albums.tracks)
}

// ----------------- End of Multiplicity Invariants -----------------
```

Table 2 Specification of state for the “Manage Multiple User Music” role and the corresponding Alloy code.

Additional constraints come directly from the diagram in table 2: for all cycles in this diagram we add invariants in the Alloy code. For example, *inv1* requires for all users, that tracks from all user playlists (for a given user) are tracks of the same user. The MultipleUserMusic Alloy module can be analyzed by means of generating different examples. We do not give them here since they are similar with examples from figures 3 and 4.

Based on the description of our small case study, we can build a state structure model of the “Manage Multiple Artist Music” role (see figure 5).

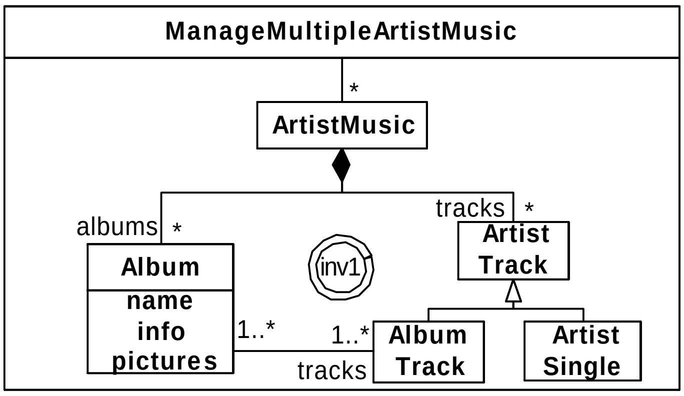


Figure 5 The state structure model for the “Manage Multiple Artist Music” role.

This model and the corresponding Alloy code are built in the same way as for the “Manage Multiple User Music” role. Therefore we do not show the Alloy code here.

Let’s conclude: up to this point we have the role state structure models of the both “Manage Multiple User Music” and “Manage Multiple Artist Music” roles. In the next section we show how a composition of these two models is done.

3.2 Composed view

As we explain in [Balabko03], the modeling process of a complex system consists in building models in specific contexts and finding identical elements in these models. In this section we continue the example of a model of a Simple Music Management System that gives an intuition of the meaning of identical elements (more details about identical elements see in [Balabko03]). Let’s imagine an external observer who perceives the work of two users (see figure 6).

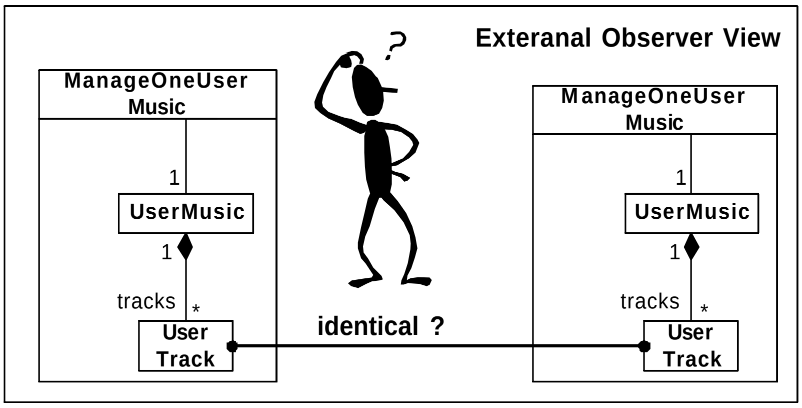


Figure 6 Identical model elements.

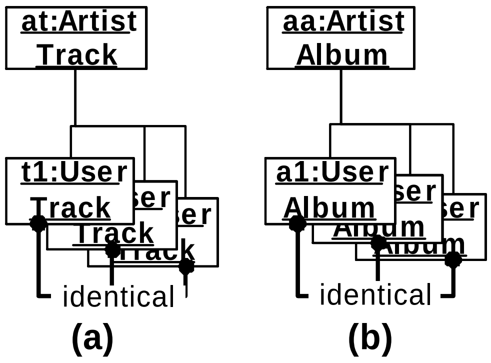
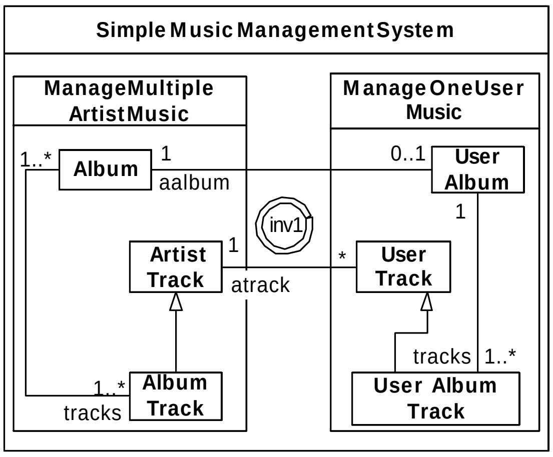


Figure 7. Modeling of identical model elements: **(a)** Identical User Tracks; **(b)** Identical User Albums

We suppose that this external observer can not see all details about user tracks: how they are created, used and deleted. The only one thing that he can observe is the content of tracks. This defines an External Observer view. From this point of view, the external observer may say that some user tracks of some different users are identical, i.e. the music of these tracks is the same. The question is, if we have to model this identity or not. In our case study, this identity is important because it takes into consideration the Artist Track concept from the “Manage Multiple Artist Music” role: two tracks can be considered identical if they correspond to the same artist track.

To model the identity of model elements in our framework, we define a new attribute (concept) that is associated with all identical elements. In our case study, this new attribute is the Artist Track attribute from the “Manage Multiple Artist Music” role (see figure 7.a). Figure 7.a shows that an Artist Track has multiple identical User Tracks. In the same way as User Tracks can be identical because they refer to the same artist track, user albums can also be identical because they refer to the same artist album. Therefore, we model an Artist Album has multiple identical User Albums (see figure 7.b).

Let’s conclude. We modeled the state structure of two roles (“Manage Multiple User Music” and “Manage Multiple Artist Music”). We also modeled how the state structures of these two roles are composed (using a diagram from figure 7). All these models give us the state structure model of a Simple Music Management System. At this point we can create an Alloy code (see table 3) that reflects the composition of the two roles and then we can do the analysis of the complete model.



```
1.  module MusicManagementSystem   //module declaration

2. open MusicManagementSystem/UserMusic/MultipleUserMusic
3.  open MusicManagementSystem/UserMusic/OneUserMusic
4.  open MusicManagementSystem/ArtistMusic/MultipleArtistMusic
5.  open MusicManagementSystem/ArtistMusic/OneArtistMusic

7.  sig UserTrack1 extends UserTrack {atrack: ArtistTrack}
8.  sig UserAlbum1 extends UserAlbum{aalbum: Album}

9.   fact { all ut:UserTrack | ut in UserTrack1 }
10. fact { all ua:UserAlbum | ua in UserAlbum1 }

// ------------------  Multiplicity Invariants -------------------------
11. fact {      // Multipl: Album (1) <- (0..1) UserTrack
  all  um:UserMusic  |  all  a:Album  |  sole  ua:um.albums  |  a  =
ua.aalbum }
// ------------------  End of Multiplicity Invariants ----------------

// ------------------  Multiplicity Invariants -------------------------
12. fact inv1 {  all um:UserMusic | all ua:um.albums |
                    ua.aalbum.tracks = ua.tracks.atrack }
// ------------------  End of Multiplicity Invariants ----------------------

13.  fact {          // Definition of UserSingle: a user track
                     // that is not a part of any user album
   all um:UserMusic | all us:UserSingle |
   (us in um.tracks) => (us.atrack not in um.albums.tracks.atrack) }
```

Table 3 State Structure Composition for the “Manage Multiple User Music” and “Manage Multiple Artist” roles.

We have to reflect in the Alloy code new relations between the AtristTracks and UserTrack types and between the Album and UserAlbumTrack types. To make a new relation between two already specified attributes in the current version of Alloy, we have to extend existing types: UserTrack and UserAlbum (lines 7-8 in table 3). To ensure that new attributes (UserTrack1 and UserAlbum1) and their predecessors participate in the same relations we created two facts (line 9-10 in table 3). Line 11 in table 3 specifies the multiplicity invariant. Note that this multiplicity is specified in the context of one user (as it is shown in a diagram from table 3): we require that for any artist album there is only one user album in the context of a given user music. When we create an Alloy code, we have to take into account possible “conceptual cycles”. Like in previous examples we have to create invariants for these cycles.

The last constraint[2] that we add to the Alloy code is related with the definition of a user single track: a user track that is not part of any other user album (see introduction). The fact from line 13 specifies this constraint. It tells that for any user single of some given user, the artist track for this user single is not an artist track for any album track for the same user.

3.3 Analysis of the Composed Model

Based on the Alloy code from table 1, several instance diagrams can be generated. One of these diagrams is shown in figure 8.

[2] This constraint was discovered in the result of the analysis of instance diagrams generated by the Alloy Constraint Analyzer.

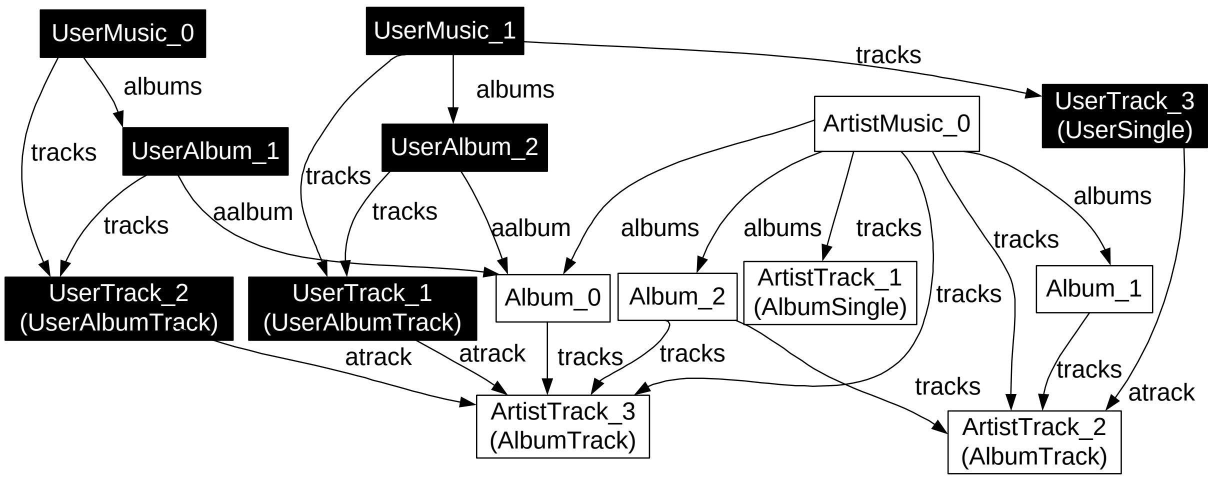


Figure 8 Instance diagram for the Simple Music Management System.

We used filters provided by the Alloy Constraint Analyzer tool to hide instances of the User Play List attribute. This attribute is not involved in the composition of base roles and therefore is not necessary for the analysis of the composition.

To make the analysis easier we colored the instances of attributes from the “Manage Multiple User Music” role in black. All other instances of attributes are from the “Manage Multiple Artist Music” role. The diagram from figure 8 can be used for the analysis of the model of a Simple Music Management System. This analysis can be done in collaboration with a customer who asked for the development of this system. The goal of the analysis is to show possible states of the system (instance diagrams) and ask if these states can adequately represent the reality. For example, the following question can be asked: If a user single track (see UserTrack_3 in figure 8) can be related with an artist album track (see ArtistTrack_2 in figure 8), i.e. if a user can add to its music collection only one track from an artist album versus adding the whole album? Similar questions can be asked based on several instance diagrams like one in figure 8. This kind of model analysis helps for avoiding mistakes that can be discovered later in the implementation phase.

4 Conclusion

In this work we presented the small case study of a Simple Music Management System. This case study explains how our approach can be used for the modeling of role state structure, a composition of role state structures and its analysis. There are several approaches that allow for the composition of roles.

Many of these approaches are very implementation oriented. Michael VanHilst and David Notkin describe a method for the implementation of roles in C++ [VanHilst96]. Several works consider role composition using implementations with role wrappers. Wrappers can intercept incoming or outgoing messages from different roles and impose rules on message passing between these roles (a communication protocol). For example, M. Aksit specifies a communication protocol between roles with Abstract Communication Types [Aksit94], L. Andrade and J. Fiadeiro specify it with coordination contracts [Andrade01], M. Kandé specifies it as a connector that defines a pattern of interactions between two or more components [Kande00]. Achermann, F., et al. in [Achermann01] propose a more general approach for composition that is based on a new composition language, PICCOLA, that can be used as unique language for components composition. This language can provide support for key concepts form the various existing composition languages like: scripting languages (Perl, Python), coordination languages (Linda, Manifold), architecture description languages (Wright, Rapide), and glue languages (Smalltalk)

However, none of these approaches can be used by an analyst whose goal is to build system requirements without looking at the implementation details. All these composition approaches are quite difficult to use and do not have visual representation that is more convenient for requirements engineer. In the field of requirements engineering the following approaches based on role modeling can be used: RIN – Role Interaction Networks [Singh92], RAD – Role Activity Diagram [Ould95] and OORAM – the Object-Oriented Role Analysis Method [Reenskaug96]. These three approaches are quite similar. Roles are considered as sets of sequentially ordered actions and/or interactions. The main drawback of these approaches is that goals are difficult to model with these PMTs. Another problem with these PMTs is that states are defined in such a way that it is difficult to split the state into subsets (for different contexts). These two problems are related to the fact that the PMTs do not have a state structure (state is considered as an instant between connective actions). We have not seen many approaches that allow for the composition and analysis of object state structures. One similar approach is the View based UML (VUML) described in [Nassar03]. This approach provides the concept of a multiview class that specifies the composition of models from the point of view of different system users. The dependencies between views are specified using OCL. However the VUML approach does not provide means for the evaluation of a composition before its implementation.

The main contribution of our work is the possibility for the formal visual analysis of a composed model. The analysis of a composed model allows for reaching an agreement with a customer that a model is adequate before a system get implemented.

5 References

[Achermann01] Achermann, F., et al., Piccola - a Small Composition Language, in Formal Methods for Distributed Processing - A Survey of Object-Oriented Approaches, H. Bowman and J. Derrick, Editors. 2001, Cambridge University Press. p. 403--426.

[Aksit94] Aksit, M., et al., Abstracting Object Interactions Using Composition Filters, in Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming, R. Guerraoui, O. Nierstrasz, and M. Riveill, Editors., 1994, Springer-Verlag. p. 152--184.

[Andrade01] Andrade, L. and J. Fiadeiro. Coordination Technologies for Managing Information System Evolution, in Proceedings of CAiSE'01, 2001, Interlaken, Switzerland: Springer-Verlag.

[Balabko03] Balabko, P., Wegmann, A., “A Synthesis of Business Role Models”, in Proceedings of ICEIS 2003, ICEIS Press, Anger, France, download from http://lamswww.epfl.ch/publication/lams_publication_selection.asp

[Chang99] Chang, S.K., et al. “The Future of Visual Languages”, in Proceedings of IEEE Symposium on Visual Languages, 1999, Tokyo, Japan. pp. 58-61

[D'Souza98] D'Souza, D.F. and A.C. Wills, Objects, Components, and Frameworks With Uml: The Catalysis Approach. Addison-Wesley Object Technology Series. 1998: Addison-Wesley Pub Co.

[ISO96] ISO/IEC, (1996). 10746-1, 3, 4 | ITU-T Recommendation X.902, Open Distributed Processing - Basic Reference Model - Part 2: Foundations.

[Jackson02] Daniel Jackson, “Micromodels of Software: Lightweight Modelling and Analysis with Alloy”, Software Design Group, MIT Lab for Computer Science, 2002, downloaded from http://sdg.lcs.mit.edu/alloy/reference-manual.pdf

[Kande00] Kandé, M.M. and A. Strohmeier. Towards a UML Profile for Software Architecture. in UML'2000 - The Unified Modeling Language: Advancing the Standard. 2000. York, UK,: LNCS Lecture Notes in Computer Science.

[Nassar03] Mahmoud Nassar, at al. “Towards a View Based Unified Modeling Language”, in Proceedings of ICEIS 2003, ICEIS Press, Anger, France.

[Naumenko02] Triune Continuum Paradigm: a paradigm for General System Modeling and its applications for UML and RM-ODP, Ph.D thesis number 2581, EPFL June 2002.

[Ould95] Ould M. A., Business Processes: Modeling and analysis for re-engineering and improvement, John Wiley & Sons, 1995

[Reenskaug96] Reenskaug, T., et al., Working With Objects: The OOram Software Engineering Method. 1996 ed: Manning Publication Co

[Singh92] Singh, B. and G.L. Rein, Role Interaction Nets (RINs): A Process Description Formalism, 1992, MCC: Austin, TX, USA, Technical Report CT-083-92

[VanHilst96] VanHilst, M. and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. in Proceedings of OOPSLA'96. 1996: ACM Press.