Formalizing Metamodel of Requirements Management System

[1]D.S. Kildishev <kildishev@ispras.ru>
[1 2 3 4]A.V. Khoroshilov <khoroshilov@ispras.ru>
[1] Ivannikov Institute for System Programming of RAS,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[3] Moscow Institute of Physics and Technology,
9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia
[4] Higher School of Economics.
20, Myasnitskaya Ulitsa, Moscow 101000, Russia

Abstract. Requirements play an important role in the process of safety critical software development. To achieve reasonable quality and cost ratio a tool support for requirements management is required. The paper presents a formal definition of a metamodel that is used as a basis of Requality requirements management tool. An experience of implementation of the metamodel is discussed.

Keywords: Requirement, model, requirements management

DOI: 10.15514/ISPRAS-2018-30(5)-10

For citation: Kildishev D.S., Khoroshilov A.V. Formalizing metamodel of Requirements Management System. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018, pp. 163-176. DOI: 10.15514/ISPRAS-2018-30(5)-10

1. Introduction

The development of complex systems is always a sophisticated task. The development of complex safety-critical systems, where the cost of errors is especially high, is particularly complicated. Modern best practices suggest that precise and accurate requirements management is an important element to solve that task.

Requirements managements in the context of safety-critical system development include the following aspects:

- building a catalogue of requirements;
- traceability links to sources of requirements;

- traceability links from other development artefacts like tests and code to requirements;
- configuration and version management;
- change management including change impact analysis.

The paper presents a formal definition of a metamodel that is used as a basis of Requality requirements management tool that is aimed to cover all the aspects. Implementation details of the metamodel in the tool are also discussed and future directions are considered.

2. Related works

The problem of requirements management is not a new one. This activity was known as a very important one for years. As an example we may cite a publication from 1997:

"The inability to produce complete, correct, and unambiguous software requirements is still considered the major cause of software failure today" [1].

But the requirements engineering task is still the subject of different investigations. Some of them defines a methodology [2], a model [3] or a framework [4]. Also, there are papers presenting development story of some tools, like [5].

Some papers describe both requirements model and its application in a specific tool. For example [6] designs a tool for management of requirements in form of specific models or [7] that defines some details about a feature management tool for product lines. Another paper [8] defines requirements as constraints and examine core concepts related to its implementation in a real tool.

There are many commercial requirements management tools with a little information about architecture and implementation details. There only a few open source tools are known and cited in publications like ProR [9] or ReqLine [10].

None of the papers on the tools discusses its core model in a formal way. Some approaches and models are listed in [11] but it specifies mostly methodological aspects.

3. Base model

3.1 Preliminaries

The process of software development can be made in different ways. There are some general views on requirements management tool's functions but the set of requirements for this tool in specific areas may be different.

One of the ways to deal with such problem is to develop a model for that tool. This approach can be found in [7] or [8]. The model helps to define core concepts of the tool and prove some theorems over its functions.

We need to provide some terminology before starting a model. First, we will define what the *requirement* is. In this paper, *requirement* means a limitation or definition of some system's or component's functional. For our model *requirements* are unique

objects that may have a specific description written by natural language and are placed in some tree structure defined below.

3.2 Base model

Definition 1. A *tree G* is a triple (V, E, r_0), where:

- V - a set of vertices.
- $E \subset V \times V$ - a set of edges that is an asymmetrical relation on V.
- $r_0 \in V$ - a root of the tree.
- There are no incoming edges for r_0 and there are no more than one incoming edge for the other vertices.
- All vertices are reachable from r_0.

If (v_1, v_2) ∈ E then v_1 is denoted as a parent(v_2), while v_2 is called a child of v_1. We define relation $\text{reachable}_E(v_1, v_2)$ as a transitive and reflexive closure of the relation E.

Definition 2. *Attributed tree* AT = (G, Key, Value, attrs) consists of:

- a tree G = (V, E, r_0);
- a set of attribute keys Key;
- a set of attribute values Value;
- a functional relation attrs: V → (Key → Value) that provides each vertex with a set of attributes.

A set of all possible attributed trees is denoted as ATrees.

An attributed tree is a convenient framework to represent requirements [12] with the following semantics. If a vertex v ∈ V represents a requirement for a target system and there are children v_1, … v_n of v, then the children represent a decomposition of the requirement v. In other words, if a system satisfies to requirement v then it satisfies to all requirements v_1, … v_n and vice versa.

Attributes of vertices contain various information about the requirements, for example a unique identifier, description of the requirements in natural language, its representation in a formal notation, version, etc.

An interesting particular case is the attributes, whose value is a vertex v ∈ V or a set of vertices vs ⊆ V. It allows to define and to manage relations between different vertices. For example, such attributes can be used to represent traceability links between high level and low level requirements. Formally, this case is achieved if V ∪ $\wp$(V) ⊆ Value.

4. Declarative model

4.1 The extension of the base model

The base model of requirements catalogue is an attributed tree, where each requirement has a particular set of attributes. This model is convenient for analysis of the

catalogue, e.g. for formal analysis, analysis of test coverage, traceability analysis, etc. At the same time, it is difficult to manage such model manually because there are usually many interdependencies between elements and its attributes. Here and after term vertex (element of set V) and elements of requirements catalogue are used interchangeably.

That is why we introduce a declarative model of requirements catalogue that allows us to automate the handling of such dependencies. The purpose of the declarative model is to store requirements catalogue in more compact and manageable way.

The declarative model is defined stepwise. Each step is accompanied by definition of the transformation of the declarative model to the raw basic model.

4.2 Predicates

If requirements are developed for a product line, there is a number of requirements shared between different variations of the product. A natural wish is to have a single requirements catalogue for the product line and the ability to build a specific one for a particular version of the product. That means there is a need to delete a subset of requirements from the catalogue if the subset is not applicable to the target product.

The similar situation happens when a catalogue is used to represent requirements of several revisions of a standard or to represent requirements of a standard with optional elements.

To introduce such ability we propose to choose especial key predicate ∈ Key, whose values are boolean. If an element has attribute predicate with value false, this element and all its children are removed from the catalogue during transformation.

The first declarative model DM_1 is an attributed tree ((V, E, r_0), Key ⊔ {predicate}, Value, pattrs) that is transformed to the base model ((V', E', r_0), Key, Value, attrs) according the following rules:

- V' = {v ∈ V: ∀ v' ∈ V reachable_E(v',v)
  predicate ∉ pattrs(v') ∨ pattrs(v')(predicate) ≠ false };
- E' = E ∩ (V' × V');
- ∀ v∈V' attrs(v) = {(k,val)∈pattrs(v): k ≠ predicate}

4.3 Calculated attributes

It is an often situation when attribute value depends on values of the other attributes of the same element or even on attributes of the other elements. To express such dependencies explicitly we propose the second declarative model DM_2 that is an attributed tree (G, Key, FValue, fattrs), where

- FValue = Func × Value;
- Func = ATrees × V × Key × Value → Value.

The declarative model DM_1 corresponding to the model DM_2 is an attributed tree (G, Key, Value, attrs):

∀ v ∈ V (k,val) ∈ attrs(v) iff

∃ (k, (func,fval)) ∈ fattrs(v): val = func(AT, v, k, fval)

To build such requirements model it is required to solve a set of equations defined by fattrs. A simple approach is to apply fixed point iteration, while some additional implementation details will be considered in section V. There are declarative models that define a set of equations with no solutions or with non-unique solutions. A simple but reasonable limitation that allows avoiding such models is a prohibition of cyclic dependencies between attributes.

A particular case when an attribute has a constant value val is represented in the declarative model DM_2 as a pair (prj_4, val), where prj_4 is a projection function by the fourth argument: prj_4(AT, v, k, val) = val. Please note that in DM_2 predicate is considered as a regular element of the set Key.

4.4 Attribute scope

Another often situation happens when an attribute is applicable to the whole subtree and it has the same value for all elements. Or a similar case is when an attribute is applicable to all children of the particular element.

To handle such situations we propose the third declarative model DM_3 that is an attributed tree (G, Key, SValue, sattrs), where SValue = FValue × Scope, Scope = {S_L, S_{DC}. S_S} with an element having the following semantics:

- S_L – an attribute is available only in the element where it is defined.
- S_{DC} – an attribute is available in the element where it is defined and in all its direct children.
- S_S – an attribute is available in the element where it is defined and in all its successors.

An example of attribute scope can be seen on Fig. 1. White rectangles are Vs. Arrows mean child-parent relation. Attribute with some scope is defined in r_0. Grey rectangles represent different possible scopes of A and the subtrees where it will be accessible.

\A transformation of declarative model DM_3 to the model DM_2 is straightforward: DM_2 is an attributed tree (G, Key, FValue, fattrs), where fattrs(v) = {k → fval} such that

- (1) {k → (fval, anyscope)} ∈ sattrs(v)
- (2) {k → (fval, S_{DC})} ∈ sattrs(parent(v)) if rule (1) is not applicable,
- (3) {k → (fval, S_S)} ∈ sattrs(v') if rules (1) and (2) are not applicable ∧ $reachable_E$(v',v) ∧

∀ v" ∈ V ∀ val ∈ Value ($reachable_E$(v",v) ∧ $reachable_E$(v',v")) ⇒ {k → (val, S_S)} ∉ sattrs(v").

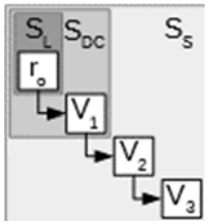

Fig. 1. Attribute scopes

It is interesting to note that nonconstant scoped attributes can get different values in different elements because its function can depend on the vertex as a third argument.

4.5 Reuse of subtrees

The next item to consider is a situation when there are several subtrees of requirements that are very similar each other up to some limited number of details. In this case, it would be ideal to have a single copy of the subtree and the ability to clone it with some modifications. This approach is usually called reuse [13].

The fourth declarative model DM_4 is an attributed tree ((V, E, r_0), Key ⊔ {cp}, SValue, cpattrs) with especial key cp that satisfies the following constraints:

- ∀ v ∈ V∀ value ∈ Value∀ s ∈ Scope cp ∈ cpattrs(v) ∧ cpattrs(v)(cp) = ((prj_4,val), s) ⇒ val∈ V ∧ ∀ v' ∈ V (v, v') ∉ E;
- E ∪ {(v, cp(v))| v ∈ CC(DM_4)} does not contain loops, where CC(DM_4) = {v ∈ V| cp ∈ cpattrs(v)} and cp(v) - val∈ V from the constraint above.

The transformation of the model DM_4 to the model DM_3 = ((V', E', r_0), Key, SValue, sattrs) is performed by the following algorithm:

1. $curDM_4$:= DM_4
2. If CC($curDM_4$) is empty, take DM_3 = $curDM_4$ with removing cp from the Key set and finish.
3. Let $curDM_4$ is ((V, E, r_0), Key ⊔ {cp}, SValue, cpattrs).
4. Choose any v_0 ∈ CC($curDM_4$) such that ∄ v ∈ CC($curDM_4$) $reachable_E$(cp(v_0),v). Existence of such element follows from lack of loops in E ∪ {(v, cp(v))| v ∈ CC(DM_4)}.
5. Assume without loss of generality ∀ v∈V (v_0,v)∉V.
6. Build newDM_4 = ((V', E', r_0), Key ⊔ {cp}, SValue, cpattrs'), where
   - V' = V ∪ {(v_0,v') | v'∈V ∧ $reachable_E$(cp(v_0),v') }
   - E' = E ∪ { (v_0,(v0,cp(v_0))) } ∪ { ((v_0,v'), (v_0,v")) | (v',v") ∈ E ∧ $reachable_E$(cp(v_0),v')}
   - ∀ v ∈ V\{v_0} cpattrs'(v) = cpattrs(v)
   - cpattrs'(v_0) = {(k,val) ∈ cpattrs(v_0): k ≠ cp}

- cpattrs'((v_0,v')) = cpattrs(v')

Please note that $newDM_4$ satisfies both constraints of the fourth declarative model.

7. $curDM_4$:= $newDM_4$ and goto step 2.

Lemma 1. The algorithm terminates for any DM4 satisfying the constraints.

The proof is based on the fact that the cardinality of $CC(curDM_4)$ is decreased every iteration because of the choice of the v_0 at step 4 that guarantees that elements with attribute cp are not cloned, while one such element loses that attribute.

Lemma 2. The result of transformation does not depend on the order of the selection of elements at step 4.

The idea of the proof is that transformations that can be chosen in non-deterministic order make modifications in non-intersecting subtrees.

Interesting to note that combination of reuse and predicate transformation can be used to define a generic subtree that is instantiated several times with different arguments using reuse transformation and the original generic subtree is eliminated with predicate transformation. Also, predicate transformation can be useful to eliminate unneeded elements from the cloned subtrees.

5. Implementation details

5.1 Identification

One of the important aspects of requirements management is requirements identification. One of the common approaches it to assign a unique identifier to each object, for example, some number or string.

In addition to that it is possible to provide each element with a qualified identifier QID defined recursively on top of identifiers ID that are unique within children of the same parent: r_0 has QID = '/ID', child v has QID = 'QID(parent)/ID'.

Let us take some example of requirements for some system. If we use QID we can have a human-readable path for each requirement. For example, we may have an element with QID = "Functional requirements/Ports/req001". As seen from the path it has a parent "Functional requirements/Ports/" and its ID is req001.

5.2 Calculated attributes

There are two objects related to attributes in the implementation. The first one, *attribute definition* A_DEF represents a pair (func,fval) from the formal declarative mode, where func is of type ATrees × V × Key × Value → Value. The second one, *attribute* A, represents a value of the attribute in the base model. A_DEF is used to calculate an actual value A when it is required.

There are several kinds of functions supported in attribute definitions.

The first kind is the *constant* functions prj_4 that always returns fval value stored in attribute definitions.

The second kind is *template functions* that stores in fval value a string with parameters encoded in curly brackets, e.g. "Hello, {K}". The value of the parameters to be used for substitution is taken from attribute with the encoded name, 'K' in the example above, of the same element.

The third kind is *formula value generator* that stores in fval value a string with an expression in a subset of JavaScript language that has access to attributes of the same element.

The fourth kind is *virtual attributes* that are implemented in Java. They have no stored fval value at all, but they have access to the whole context of the element including the complete attributed tree.

For example, Label attribute can take value of user-defined Name attribute if there is one or return system-defined identifier otherwise. Another example could be QID that calculates qualified identifier of the element as a concatenation via '/' of parent's QID with a Name of the target element.

An important additional information that the tool is able extract from attribute definition is a set of attribute keys which values are required to calculate the actual value for the given attribute by the corresponding function.

5.3 Attributes life-cycle

For each attribute stored data includes function kind and fval. The pair (funckind,fval) is denoted A_ST. System-defined virtual attributes have no stored data, they are added to elements on the fly.

Let us describe a common process of attributes loading for some requirement.

1. Set of A_ST is loaded from storage to A_DEFS.
2. Set of scoped attributes that are applicable to the target one is taken from its parent and is added to A_DEFS.
3. The A_DEFS set is handled by Attribute_Calculation procedure described below.

If attributes are changed by the user using GUI session, the tool has the same A_DEFS set that contains a subset of changed attribute definitions. Then the tool applies the same Attribute_Calculation procedure as follows.

1. A_DEFS set is extended with attributes of the target element that depends on any attribute already belonging to A_DEFS.
2. The order of evaluation of attributes from A_DEFS is calculated. The order can be defined as ORDER = (K_1..K_n) where K_i is the key of the attribute.

-$\forall Ki,Kj \in ORDER$ if K_j depends on K_i then $i<j$.

The algorithm is described in the next section.

3. For each A_DEF in the A_DEFS value of A is calculated and placed to AS.

After this procedure AS contains an actual state of attributes after provided changes.

5.4 Order extraction algorithm

As an input of order extraction, we have KEYS = (K_1.. K_n) that is set of attributes name in some random order and DEPS = ($K_i \rightarrow (K_{j1}..K_{jm})$) - a map of attributes dependencies. The algorithm is as follows:

1. ORDER is set to empty collection.
2. OSET is the set of handling nodes.
3. Extract revert dependencies DEPS_R. DEPS_K=($K_j \rightarrow (K_{i1}.. K_{il})$). If K_i depends on K_j then DEPS_K contains $K_i \rightarrow K_j$ record.
4. Place KEYS to OSET.
5. Set flag MOD is to False.
6. In OSET look for candidate KK with DEPS_K[KK] = KSET that complies one of following rules:
   - KSET is empty.

OR

   - !$\exists K_i \in$KSET: Ki$\in$ OSET.
7. If K_K was found then:
   1. MOD set to True.
   2. K_K removed from OSET.
   3. K_K added to ORDER.
8. If MOD = True & |SET|!= 0 then go to step 4.

At the end of execution, the ORDER will contain the order in which A's values calculation.

5.5 Attribute change management

The introduction of scope and calculated attributes requires the management of attributes changes to keep all dependent attributes up-to-date.

There are two possible strategies to deal with attribute updates. The first approach is to commit all changes at runtime. The second one is to collect changes in AS and then apply them all by request. Immediate commit is tending to be simpler but more computing - intensive. Late updates require fewer calculations but need more memory. For our tool, we use the second approach because we have large catalogues with a possibility of complex relationships between its elements.

Late changes can be defined in form of new object — changes set CS=(K $\rightarrow$ OP, K$\rightarrow$ A_{OLD}, K$\rightarrow$ A_{NEW}) where K is the key of attribute, OP $\in$ (remove, create, modify) is the operation over attribute, A_{OLD} is the value of attribute in AS before operation, A_{NEW} is the new attribute value after operation.

For attribute changes change set needs to store A_DEFS, so minimal CS = (K, K$\rightarrow$ A_{OLD}, K$\rightarrow$ A_{NEW}). To use these changes set we need to extend the model of attributes set of A.

When all attribute modifications are collected we need to apply all that changes to calculate actual values of attributes. It is implemented in the same way as it was described in section V.C.

One more problem with attribute changes is that some of the changes need to be propagated from one requirement to another. To deal with this problem we define a concept of *change propagators*. If A_DEF (virtual attributes only for now) depends on attributes from the external element it registers a function-change propagator that is called when some change set is applied to attributes of that element. The change propagator evaluates if the changes impact the target attribute and initiate its recalculation if it is required.

5.6 Lazy loading

When we speak about a model of requirements in some common application like avionics we need to take into account the number of distinct requirement. Sometimes the number of artefacts for such models tends to be in the thousands or tens thousands. In that case, direct management of requirements may require a lot of resources.

To solve this problem we use the lazy loading principle. That means that AT will contain only those Vs that are requested during the usage of the model. In most cases that means that in G we have a subtree $G_L \in G$ that contains r_0 and some subtrees that are used during the current working session.

But laziness of model leads to some difficulties. First of all, we need to overlook AT instead of AT_L if we need to assure that V with given ID exists. This problem can be solved by caching id-related information in CacheStore that is always available.

5.7 Attribute types

In practice, the value of an attribute may have one more property – a *type*. One possible set of types includes Integer, Boolean, String, Float. Also, we may define types for Collection and Enumeration. In most cases, the value still is the simple constant.

But some attribute types cannot be defined as a single value and need to store and manage some additional data. For example, *Collection* type may use specific object LIST = (T_V, $V_1..V_n$) where T_V is the type of collection's value and ($V_1..V_n$) are the values stored in the collection.

One more specific type is *Enumeration*. First, enumeration requires definitions of its values. It can be made by means of ENUM_DEF = (V_T, $V_1...V_n$) that is similar to LIST one. But to define an attribute with one selected enumeration value we need to define one more object ENUM = (K_B, V_S) where K_B is the key of A with ENUM_DEF and V_S is the selected value. But in a case we introduce an ENUM, we need to ensure that for every ENUM we will have an A_D where T = ENUM_DEF and V_D will contain V_S.

5.8 References

One more problem is the implementation of relations between elements of the catalogue. Some tools manage them as the set of specific objects placed in the distinct set.

In our model relations are presented in form of specific attribute type REFERENCE. For this type we introduce value object REF_VALUE (REF, V, ERR) where REF is a string that can be resolved to V, usually containing some kind of identifier, V is the corresponding element if there is any matched by identifier, ERR is a string with an error message if REF cannot be resolved or contains incorrect value.

In this case, REF_VALUE initially contains only REF field. If someone requires the result of REF_VALUE resolution then the tool tries to resolve the REF and then fills V or ERR.

References are also required some additional handling to support its consistency. In a case REF or target V is changed we may need to track its changes and update related REF_VALUE.

One more specific problem is reverted links. If we have a relation $V_1 \rightarrow V_2$ we may need to know for V_2 that it has a relation to V_1. This kind of relations is called "reverse references".

If links are stored in AT then we may use one more function $(V_2, LN) \rightarrow V_1$ to store reverse relations. If we define a new type of attributes or the specific state of REF_VALUE then we face a problem of keeping it up-to-date.

In our model, we store reverts links in the cache in form of $(V_2, LN) \rightarrow (V_1 \dots V_n)$ function. That allows us to easily get revert links on V_2 if the state of cache is valid.

In a case of completely loaded AT the problem is not so difficult to solve because we always have the actual state of every V. But we cannot guaranty the V's state in case of a partially loaded AT that happens in case of lazy loading.

If we have some loaded $AT_L \subseteq AT$, relation $(V_1, LN) \rightarrow V_2$, $V_1 \notin AT \wedge V_2 \in AT$ then if we need to get revert links on V_2 we may need to load the whole AT to be sure that all possible V_1 were found.

In our case, this problem is solved by storing reverse links in the cache. But in this case, we still have one necessary problem. Let us introduce some link $L(V_1, V_2, LN)$. If we already resolve this link then the record in cache tends to be present. But what if we introduce V_2 in the model when V_1 is loaded and the link is resolved was not found? The situation takes place when V_2 is loaded by the lazy method, created or modified.

In the worst case, we need to track changes of the whole AT for all links. A better solution is to manage some kind of scope for which link tends to be resolved. That is not implemented yet, but it is in our plans.

Relations can be used for some specific activities. One of them is changes management. Changes management is performed when some V_1 with links $(L_1..L_n)$ is changed. In this case, some operations will be performed on V's obtained from $L_1..L_n$. The nature of such operation can be different. For some tools, those Vs will be marked

in a model with the specific flag. In other cases, the models can define additional actions depending on the kind of change.

Conclusion

We presented a formal metamodel that is used as a basis for building Requality requirements management tool. We covered different difficulties related to its implementation. But the experience demonstrates that the model allows handling quite big requirements catalogue with many relations between its elements.

The future work includes analysis and implementation of new kinds of functions for calculated values and development of user-friendly patterns for solving common user tasks on top of the semantics defined in the paper. Another direction is research of possible compositions of the formal model provided by the tool and formal models used to represent particular requirements.

Acknowledgment

This study was supported by RFBR grant #17-07-00734.

References

[1]. R. Thayer. M. Dorfman. Software Engineering. IEEE Computer Society press, 1997., 552 p.

[2]. M. Palumbo. Requirements Management for Safety Critical Systems. Available: http://www.railwaysignalling.eu/wp-content/uploads/2015/06/Req_mgt_safety_critical_system_Mpalumbo.pdf. Accessed: 3-Apr-2018

[3]. P. Roques. Modeling Requirements with SysML. Requirements Engineering Magazine, issue 2015-02, 2015.

[4]. Open Group Standard. Dependability through Assuredness (O-DA) Framework. The Open Group Releases, 2013.

[5]. A. Nordin, A. Ikhwan Omar, M. Usamah Megat Mohamed Amin, N. Salleh. .Development of scenario management and requirements tool (SMaRT): towards supporting scenario-based requirements engineering methodology. International Journal of Engineering & Technology, Vol 7, No 2.14, Special Issue 14, 2018, pp 62-65.

[6]. D. Lozhkina, S. Staroletov. An online tool for requirements engineering, modeling and verification of distributed software based on the MDD approach. In Preliminary Proceedings of the 11th Spring/Summer Young Researchers'Colloquium on Software Engineering, 2017, pp. 23-28.

[7]. T. von der Maßen, H. Lichter. RequiLine: A Requirements Engineering Tool for Software Product Lines, Software Product-Family Engineering, 2003, Heidelberg, pp. 168-180.

[8]. N. W. Mogk. A Requirements Management System based on an Optimization Model of the Design Process. In Proc. of the Conference on Systems Engineering Research (CSER 2014), 2014, pp 21-22

[9]. ProR Requirement Engineering Platform. [Online]. http://www.eclipse.org/rmf/pror/. Accessed: 2-Apr-2018.

[10]. ReqLine Download (ReqLine.exe). [Online]. Available: http://downloads.informer.com/reqline/. Accessed: 3-Apr-2018.

[11]. S. Hallerstede, M. Jastram, L. Ladenberger. A method and tool for tracing requirements into specifications. Science of Computer Programming, vol. 82, 2014, pp. 2–21.
[12]. Alexey Khoroshilov. On formalization of operating systems behaviour verification. In Proceedings of 11th International Conference on Computer Science and Information Technologies (CSIT-2017), 2017, pp. 168-172. DOI:10.1109/CSITechnol.2017.8312164
[13]. W. Frakes, C. Terry. Software Reuse: Metrics and Models. ACM Computing Surveys Vol. 28, No. 2, 1996.

Формализация метамодели системы управления требованиями

[1] Д.С.Кильдишев <kildishev@ispras.ru>
[1,2,3,4] А.В.Хорошилов <khoroshilov@ispras.ru>

[1] Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1
[3]Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9
[4] Высшая школа экономики, 101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. В рамках данной статьи рассматривается **метамодель**, лежащая в основе системы управления требованиями Requality. Базовая модель представляет собой дерево, каждой вершине которого сопоставлен набор именованных и типизированных свойств. Базовая модель проста и удобна для представления семантики набора требований, но оказывается не особо пригодной для формирования и сопровождения сколько-нибудь сложных каталогов требований. Поэтому авторами вводится набор декларативных моделей, позволяющих описывать каталог требований более компактным образом. При этом семантика декларативных моделей задаётся при помощи определения трансляции в базовую модель. Эти возможности обеспечивают гибкий инструментарий для компактного описания типовых наборов требований. Также в статье рассматриваются особенности реализации представленной метамодели в системе управления требованиями Requality. В заключении предлагается исследовать комбинацию представленной модели каталога требований с формальными моделями, позволяющими описывать семантику каждого требования в отдельности.

Ключевые слова: требование; модель; управление требованиями

DOI: 10.15514/ISPRAS-2018-30(5)-10

Для цитирования: Кильдишев Д.С., Хорошилов А.В. Формализация метамодели системы управления требованиями. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 163-176 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(5)-10

Список литературы

[1]. R. Thayer. M. Dorfman. Software Engineering. IEEE Computer Society press, 1997., 552 p.
[2]. M. Palumbo. Requirements Management for Safety Critical Systems. Available: http://www.railwaysignalling.eu/wp-content/uploads/2015/06/Req_mgt_safety_critical_system_Mpalumbo.pdf. Accessed: 3-Apr-2018
[3]. P. Roques. Modeling Requirements with SysML. Requirements Engineering Magazine, issue 2015-02, 2015.
[4]. Open Group Standard. Dependability through Assuredness (O-DA) Framework. The Open Group Releases, 2013.
[5]. A. Nordin, A. Ikhwan Omar, M. Usamah Megat Mohamed Amin, N. Salleh. .Development of scenario management and requirements tool (SMaRT): towards supporting scenario-based requirements engineering methodology. International Journal of Engineering & Technology, Vol 7, No 2.14, Special Issue 14, 2018, pp 62-65.
[6]. D. Lozhkina, S. Staroletov. An online tool for requirements engineering, modeling and verification of distributed software based on the MDD approach. In Preliminary Proceedings of the 11th Spring/Summer Young Researchers'Colloquium on Software Engineering, 2017, pp. 23-28.
[7]. T. von der Maßen, H. Lichter. RequiLine: A Requirements Engineering Tool for Software Product Lines, Software Product-Family Engineering, 2003, Heidelberg, pp. 168-180.
[8]. N. W. Mogk. A Requirements Management System based on an Optimization Model of the Design Process. In Proc. of the Conference on Systems Engineering Research (CSER 2014), 2014, pp 21-22
[9]. ProR Requirement Engineering Platform. [Online]. http://www.eclipse.org/rmf/pror/. Accessed: 2-Apr-2018.
[10]. ReqLine Download (ReqLine.exe). [Online]. Available: http://downloads.informer.com/reqline/. Accessed: 3-Apr-2018.
[11]. S. Hallerstede, M. Jastram, L. Ladenberger. A method and tool for tracing requirements into specifications. Science of Computer Programming, vol. 82, 2014, pp. 2–21.
[12]. Alexey Khoroshilov. On formalization of operating systems behaviour verification. In Proceedings of 11th International Conference on Computer Science and Information Technologies (CSIT-2017), 2017, pp. 168-172. DOI: 10.1109/CSITechnol.2017.8312164.
[13]. W. Frakes, C. Terry. Software Reuse: Metrics and Models. ACM Computing Surveys Vol. 28, No. 2, 1996.