A Semi-Automated Approach for the Co-Refinement of Requirements and Architecture Models

Matthias Barkowski, Melanie Schneider, Holger Giese, Johannes Dyck, Dalila Tamzalit, Dominique Blouin, Etienne Borde, Joost Noppen

▶ **To cite this version:**

Matthias Barkowski, Melanie Schneider, Holger Giese, Johannes Dyck, Dalila Tamzalit, et al.. A Semi-Automated Approach for the Co-Refinement of Requirements and Architecture Models. 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), Sep 2017, Lisbon, France. 10.1109/REW.2017.52 . hal-01873945

HAL Id: hal-01873945

https://hal.science/hal-01873945v1

Submitted on 27 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Semi-Automated Approach for the Co-refinement of Requirements and Architecture Models

Matthias Barkowski[1], Melanie Schneider[1],
Holger Giese[2] and Johannes Dyck[2]
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
[1]{forename.surname}@student.hpi.de
[2]{forename.surname}@hpi.de

Dalila Tamzalit
LS2N Lab, CNRS UMR 6004
Université de Nantes
Nantes, France
dalila.tamzalit@univ-nantes.fr

Dominique Blouin and Etienne Borde
LTCI Lab, Telecom ParisTech
Université Paris-Saclay
Paris, France
{forename.surname}@telecom-paristech.fr

Joost Noppen
School of Computing Sciences
University of East Anglia
Norwich, United Kingdom
J.Noppen@uea.ac.uk

***Abstract*— Requirements and architecture specifications are strongly related as the second provides a solution to a problem stated by the first. This coupling is typically realized by traceability links and maintaining such links becomes extremely difficult as both requirements and architecture specifications frequently evolve, and in particular when the architecture is refined providing an increasing level of details. In such case, not only the traceability must evolve but the requirements must be refined as well. We present a novel semi-automated approach to evolve non-functional requirements and their traceability links following system's architecture refinement in the context of design space exploration and automated code generation. The approach has been prototyped for AADL models refined with the RAMSES tool and for model transformations implemented as Story Diagrams.**

***Index Terms*—Requirements Engineering, Requirements Evolution, Non-Functional Requirements, Architecture Refinement, RAMSES, AADL, RDAL**

I. Introduction

Requirements and architectures are strongly related as the second provides a solution for the problem stated by the first. In Model-Driven Development, both requirements and architectures are often expressed as models and must be related to each other via links to support verification of requirements by the architecture. Establishing and maintaining such links is still largely performed manually nowadays and quickly becomes not manageable given the complexity and sizes of the systems we face today [1]. This is even more important in the context of safety-critical embedded systems, whose development must follow strict certification processes relying extensively on traceability. Furthermore, Non-Functional Properties (NFP) such as power consumption, memory usage, reliability, etc. are particularly important for embedded systems, which often must operate in environments with limited resources. Such NFPs are typically constrained by NF requirements, and Design Space Exploration (DSE) must be performed to produce a design that meets the NF requirements while optimizing NFPs.

During DSE before automated code generation, the architecture models may undergo several refinement steps in order to incorporate design patterns into the architecture or to add details specific to an operating system platform for which code shall be generated. In such cases, requirements assigned to the architecture may no longer be consistent with the new architecture as model elements may have been removed or added, often as a result of component split or merge. Following such architecture evolution, not only traceability links between requirements and architecture must evolve, but also the impacted requirements themselves. We call this the *co-refinement of requirements*.

Given today's large systems, automated or at least partially automated approaches are required to achieve the required scalability when co-refining requirements and architecture. It is therefore the purpose of this work to provide a semi-automatic approach for the co-evolution of requirements following automated architecture refinement. Our approach mainly consists of a scheme that for a given architecture refinement rule allows to orchestrate the application of corresponding predefined requirements refinement rules in order to preserve consistency.

The architecture refinements considered for this work are intended to preserve the functionality of the system and should only impact NFPs. Therefore, this first work focuses on NF requirements as it is crucial that the original NF requirements are also satisfied by the refined architecture. However, how requirements should be refined after architecture refinement cannot always be automatically determined and knowledge of the designer may be required. Therefore, the co-refinement scheme proposed in this work also provides guidance to the designer on what elements of the requirements should be checked to complete the refinement, thus limiting the required manual effort.

In the remainder of this paper, we first introduce our approach in section II with a simple running example. Next in section III we present the concrete implementation of the approach and its evaluation for a more complex architecture

refinement rule. In section IV, we discuss the generalization of the refinement of requirements in order to favor reuse of refinement rules. Then related work is presented in section V and finally the paper is concluded in section VI.

II. Approach

This section presents our co-refinement approach by first introducing a simplified architecture refinement rule used as running example. The focus is on the conceptual level without consideration of any specific languages used to model requirements and architectures and model transformations to implement the refinements. This is to ensure our approach remains applicable for the many languages that exist for requirements and architectures modeling (e.g. SysML [2], KAOS [3], AADL [6] etc.).

A. Running Example

Our running example consists of an architecture refinement rule that merges two periodic tasks into a single one functionally equivalent (Fig. 1). Before refinement, task *a* processes some data received at its input port and sends the computation results to its output port. The data is then received at the input port of task *b* through the port connection between the tasks. Task *b* then further processes the data and sends the result to its output port.

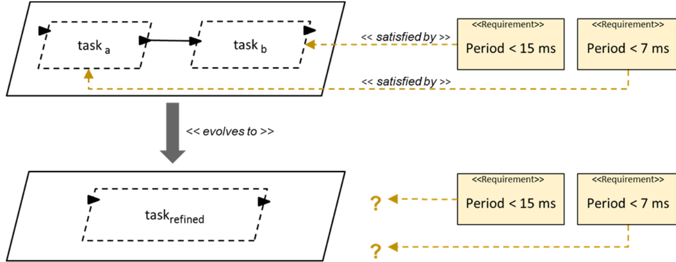

Fig. 1: Architecture refinement for the running example

After refinement, the two tasks are aggregated into a single one that performs the same function. The purpose of such refinement is to provide a single functionally equivalent component from which it will be simpler to generate implementation code.

We suppose that a requirement is assigned to each of the tasks to constrain their period (right side of Fig. 1). Each requirement contains an expression text of a given constraint language (e.g. OCL) that can be evaluated against the assigned task to check that its period property is less than a maximum allowed value. After refinement, the traceability links between the requirements and the initial tasks are broken since the tasks do not exist anymore (Fig. 1). The problem is then to refine the original requirements and to fix the assignment links and requirement's expression text. This should be performed so that the purpose of the original requirements is preserved. In this particular case, assume that the purpose of a requirement was to ensure that the input data is sampled at a sufficiently high enough rate so that every message can be received and processed. Therefore, after refinement, there should be only one requirement left, which constrains the period of the merged task to be the smallest bound of the two initial requirements (Fig. 2).

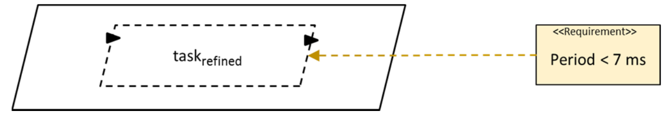

Fig. 2: Refined architecture and requirement for the running example

B. Overview of the Approach

In order to perform refinements of both the architecture and the requirements, our approach adapts an existing architecture refinement rule and extends it with a set of rules taking into account the refinement of impacted NF requirements. An overview of the approach is depicted in Fig. 3. An architecture model A_0 must be refined into a model $A_{refined}$ by application of an architecture refinement rule AR. A requirements model R_0 is linked to A_0 via assignment links between requirements and architecture model elements. After refinement, $A_{refined}$ has no allocated requirements and R_0 needs to be refined and correctly linked to $A_{refined}$.

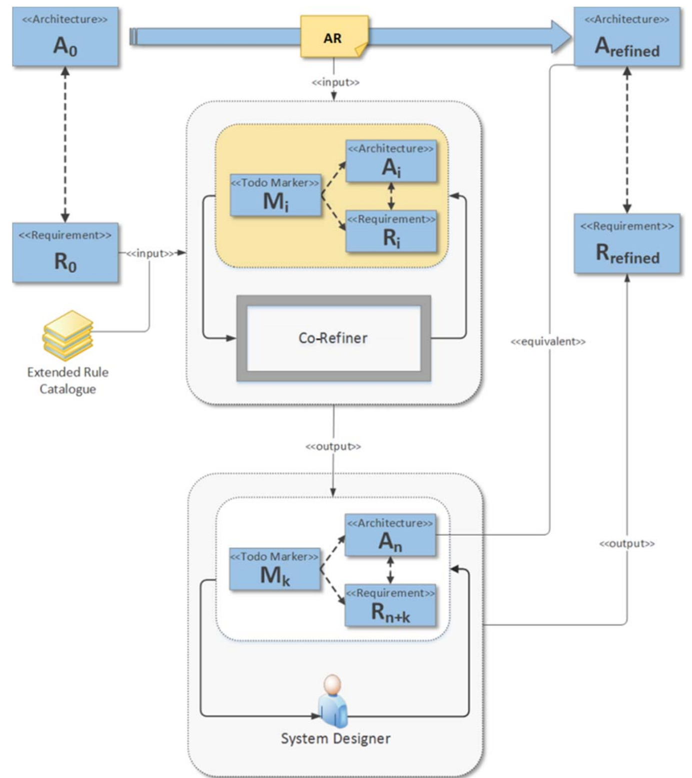

Fig. 3: Overview of the architecture and requirements co-refinement approach

For that purpose, a Co-Refiner Component (CRC) takes as input R_0, A_0 and AR and creates $A_{refined}$, $R_{refined}$ and the assignment links. The CRC has access to a catalogue of requirements refinement rules that can be executed along with a given architecture refinement rule to refine requirements.

However there are many cases where requirements can only be updated partially and may therefore be inconsistent with the refined architecture model or have broken traceability links after refinement. Such requirements will need to be processed manually in a second step. For this, the CRC produces

intermediate models of the requirements R_i and architecture A_i that are marked by a marking model M_i identifying the incompletely refined requirements and also suggesting various actions to be performed in order to complete the refinement. Such intermediate models can then be iteratively processed by the designer creating other intermediate models M_k, R_k and A_k until all partially refined requirements have been correctly refined leading to a set of refined requirements satisfying the purposes of the original requirements. The marking models can be kept to serve as a record for the refinement that occurred including traceability information between the original and refined models.

C. Co-refinement Scheme

The CRC internally applies a scheme that defines a strategy and a sequence of operations or steps for refining both requirements and architecture models given an architecture refinement rule. The order of the operations specified by the scheme ensures that all the required information from the models to be processed is available at each step. For example the deletion of model elements is postponed at the end of the scheme so that information contained in these elements is available at all steps.

The co-refinement scheme consists of 6 steps executed sequentially. In the following, we describe these steps using the running example introduced in section II.A.

Step 1: Library model elements required during later stages of the scheme are loaded.

Step 2: A match for the application condition of the given architecture refinement rule is searched within the given architecture model to be refined. If such match is found, architecture elements are marked by creating markers to store all information about the match that will be required by later steps of the scheme.

For the running example, this means finding the tasks to be aggregated, their port connection, as well as their containing process after checking that the tasks are periodic as required by the application condition of the architecture refinement rule. The found tasks and connection are then marked as being elements that will be deleted during refinement. This is depicted in Fig. 4 as a red square tag marked with an 'X'. The container of the tasks is also marked but as being modified during refinement as identified by an orange tag marked with the '~' symbol.

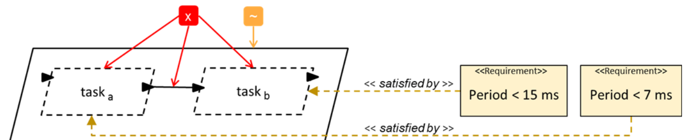

Fig. 4: Architecture markers for the running example created during step 2

Step 3: Requirements that will be impacted by applying the architecture refinement rule are searched for. This step must be performed before the architecture refinement so that the complete original architecture model is available for finding the impacted requirements. All requirements that are assigned to any element of the architecture that was marked as to be modified or deleted during step 2 are considered as impacted. Markers are then created for marking the impacted requirements. Such markers are also characterized by a *category* attribute determined from the type of the NFP constrained by the requirement and obtained by parsing the requirement's expression.

For the running example, this means finding the two requirements assigned to the two tasks and marking them as impacted as illustrated in Fig. 5 by a circular tag marked with a '~' symbol. The marker category is also determined as a '*Period*' category after identification of the period NFP constrained by the requirements.

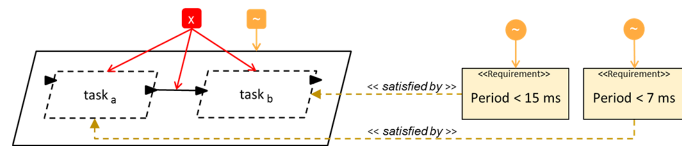

Fig. 5: Requirements markers for the running example created during step 3

Step 4: The modifying part of the architecture refinement rule is executed. However, any deletion of elements is postponed to the end of the scheme so that the elements to be deleted and that may be relevant for determining the refinement of requirements remain available. Additional architecture markers are also created during this step to mark the created architecture elements and link them to the original elements that they refine. In the end, these markers are saved and serve as a record for the refinement that occurred.

For our running example, this means creating a marker for the new aggregated task as illustrated in Fig. 6 (green tag marked with a '+' symbol) to indicate the newly created element. The marker also refers to the architecture marker for the original tasks and connection created during step 2.

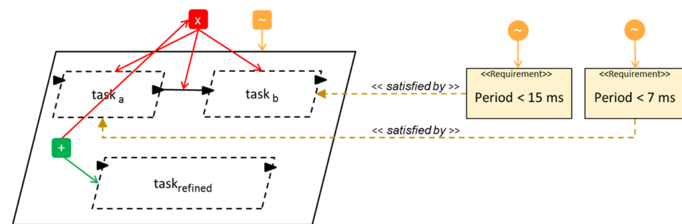

Fig. 6: Architecture refinement applied to the running example during step 4

Step 5: The impacted requirements identified during step 3 are tentatively refined using a set of requirements refinement rules provided for the given architecture refinement rule. For each impacted requirement, the requirements refinement rules are iteratively called passing to the rule the requirement and the set of architecture and requirement markers as input. Each requirement refinement rule must meet the following conditions:

- Refine the requirement and any other related requirement.
- Indicate if it was able to refine the requirement or not, so that the iteration can stop whenever a rule was able to refine the requirement.
- Mark as created any newly created requirement. Such markers must be linked to the marker of the original requirement for implementing refinement traceability.

- Flag the markers of the requirements that were refined as deleted. This ensures that a refined requirement will not be processed more than once by another rule when iterating over the impacted requirements.
- In case the rule could not refine the requirement completely, the marker of the newly created requirement(s) must be set with a *REVIEW* or *TODO* status to indicate that it must be further processed manually. In addition, the marker may indicate which specific part(s) of the requirement(s) must be checked.

In the end, each requirement that could not be refined by any rule during the iteration will have its marker set with a *TODO* status to indicate that it must be processed manually. All requirements whose marker has been marked as deleted by the refinement rules will not be deleted so that their information is preserved for recording purposes. However their assignment will be changed to the container of the previously assigned architecture elements. This provides context information on the former assignment of the requirements.

For the specific requirements of the running example, a specific rule is provided to refine the two original requirements automatically. Such rule has an application condition consisting of the following:

- The passed requirement is assigned to one and only one task of those identified by the architecture markers.
- There is another requirement that is assigned to another task that is connected via a port connection to the task of the first requirement.
- Each of such found requirements has a 'Period' category identified from its marker.
- The expression of each requirement is such that it constrains the period property of the assigned task to a *maximum* value.

When all such conditions are met, the rule creates a new requirement assigned to the merged task and having the most restrictive constraint selected among those of the 2 original requirements. For the running example of Fig. 7, this consists of the merged task having a *period < 7 ms*. A marker for this new requirement is created (green circle marked with the '+' symbol) with a status attribute set to *DONE* indicating that it does not need to be reviewed. The marker also has a link to the marker of the original requirements for refinement traceability purposes.

Alternatively, if the requirements refinement rule had not been able to determine the new requirement's expression automatically, a link from the marker to the expression would have been added to indicate that the requirement's expression needs to be inspected as indicated in Fig. 7.

Finally, the two requirements for the original tasks that are marked as deleted are updated to change their assignment to the container of the tasks. The deleted markers of such requirements also indicate that the requirements are obsolete and no longer need to be verified by the architecture.

Step 6: The deleting part of the scheme is executed removing all elements from the architecture model that were marked to be deleted by the architecture markers. The markers themselves are kept but their references to the deleted elements are transformed into attributes storing the identifier of the deleted elements, thus providing further record for the architecture refinement.

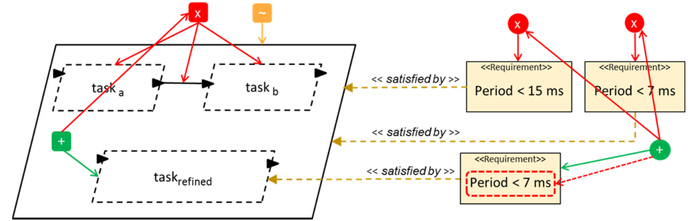

Fig. 7: Requirements refinement for the running example applied during step 5

For the running example, this consists of deleting the two original task subcomponents and their port connection (Fig. 8). The shorten arrows of the architecture marker of the deleted elements represent the attributes for the identifier of the elements.

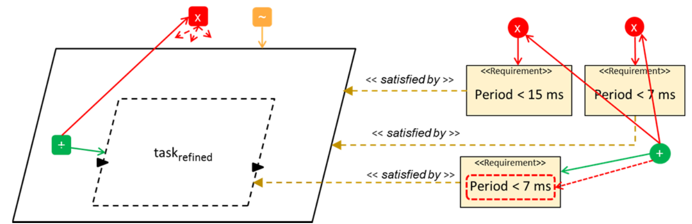

Fig. 8: Deletion of obsolete architecture elements for the running example applied during step 6

Step 7: Finally, the produced requirements, architecture, traceability and marking models are saved so that the designer can perform the required manual processing if any.

For the running example and the case where the constraint expression would not have been determined automatically, this consists of reviewing the requirement's expression and deleting the link from the requirement's marker to the expression (Fig. 9).

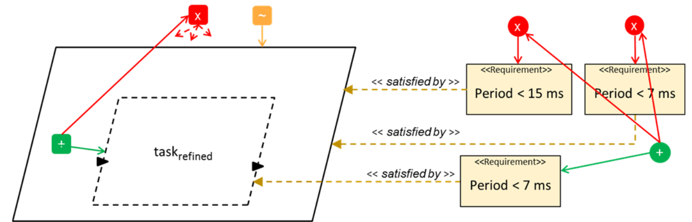

Fig. 9: Models after manual refinement

III. EVALUATION

We have presented our approach for a simple refinement rule in an abstract way, without mentioning any specific requirements and architecture modeling languages, and without considering any implementation of the described refinement transformations. In this section, we present the prototype that served to evaluate the feasibility of our approach.

A. Prototype

We used the Architecture Analysis and Design Language (AADL) [6] for modeling the system architecture and the Requirements Definition and Analysis Language (RDAL) [7] for modeling requirements. Some architecture refinement rules implemented in the RAMSES tool [4][5] have been used as case studies. The Story Diagram (SD) in-place model transformation language [8] and its tool SDM [9] were used to implement the co-refinement scheme. We briefly introduce these elements in the following.

1) Architecture Analysis and Design Language (AADL)

AADL is a rich component-based Architecture Description Language for modeling real-time embedded systems. Components are specified with type and implementation classifiers. The former describes component interaction points such as ports and bus / data accesses while the latter defines components composition as subcomponents and their interaction as connections. AADL components are divided into categories for software, execution-platform (hardware) and composite components. Software categories are *thread*, *thread group*, *data*, *process* and *subprogram* and hardware categories are *processor*, *virtual processor*, *memory*, *device*, *bus* and *virtual bus*. Composite categories are *system* and *abstract*.

The core AADL architecture language can be extended by annex sublanguages to annotate components. For instance, components behavior can be specified with the Behavior Annex (BA) [6], from which code can be generated.

2) Requirements Definition and Analysis Language (RDAL)

RDAL was designed as a requirements specification language to be used in conjunction with other languages for modeling concerns such as system architecture and use cases [10]. It also supports many well recognized RE best practices for embedded systems such as those of the FAA requirement engineering management handbook [11].

The concepts of RDAL are centered on a *contractual element* that serves as a binder of six dimensions that must be considered for the success of a project. It ensures that the typical why, what, who, when and how concerns are precisely stated. A contractual element can take several forms depending on how the contract is expressed. One well known form is a *textual contractual element*, to which a constraint written in natural or formal textual language can be associated. Such element can be assigned to model elements of the architecture upon which the constraint expression will be evaluated for contract verification.

Variants of textual contractual elements are requirements and assumptions, which are distinguished from whether they constrain the system to be developed or its environment. The expression of a requirement (or assumption) when evaluated against the assigned architecture element(s) must return a Boolean value determining if the requirement (or assumption) is verified or not.

RDAL requirements can be decomposed into sub requirements similar to goals of the KAOS language [3]. Two types of decomposition are allowed, where in the first case all child requirements of a parent requirement must be verified for the parent to be verified (AND), and for the second case several distinct refinements can represent design alternatives for which only one of the alternative needs to be verified for the parent requirement to be verified (OR). For our prototype, we only considered AND requirements decomposition with requirements expressed in the OCL constraint language.

3) Refinement of AADL Models for Synthesis of Embedded Systems (RAMSES)

RAMSES is a model transformation and code generation tool that produces C code from AADL models for ARINC653 and OSEK-compliant operating systems. RAMSES proceeds by refinements steps producing a refined version of an AADL model as an intermediate step towards code generation. The refined models include behavior annex sub-clauses that express specific components behavior resulting from the refinement. RAMSES refinement rules are currently implemented as ATL model transformations for the EMFTVM virtual machine [12]. For this work, the considered RAMSES refinement rules have been rewritten as Story Diagrams, an in-place model transformation language introduced below.

4) Story Diagrams / Story-Driven Modeling (SDM)

Story Diagrams (SD) provides a graphical syntax to describe rules for graph transformations and an interpreter for their execution. A SD consists of activities with control flow and action nodes in a similar fashion to UML activity diagrams, with the difference however that action nodes describe graph transformations. Such action nodes describe graph patterns to be matched over a model (e.g. top node of Fig. 14). Once matched, the model elements of the matched pattern can be modified or deleted and new model elements can be created (e.g. green elements in the second node of Fig. 14). SDs therefore allow for expressing complex model operations in a declarative way, leaving the complexity of matching and updating model elements to the SD interpreter. A SD can also call other SDs for reuse (Fig. 13) as well as any external Java class for further custom model processing.

B. Prototyped Rules

We prototyped our co-refinement scheme for two specific RAMSES refinement rules that were selected because they respectively consider the two different cases of splitting and merging model elements. The splitting case typically involves the addition of new elements as the number of elements after the transformation is increased while conversely, the merging case involves the deletion of model elements since the number of elements after the transformation has been reduced. Note that we do not claim that any architecture refinement can be viewed in terms of merging and splitting model elements. Covering these two cases is not sufficient to ensure our scheme can be applied to any architecture refinement. However these two very different refinement cases form a relevant starting point for evaluation.

Besides, other architecture refinement rules were also considered at a conceptual level only for further validation of our scheme, but this work is not presented here due to the lack of space. In any case, the co-refinement scheme proposed by this work is agnostic of the specificities of a refinement as its purpose is only to relate an architecture refinement rule to the

required requirements refinement rule(s) that must be applied to preserve the initial architecture and requirements consistency.

The running example of section II.A for the merging case actually consists of a simplification of a RAMSES refinement rule named *Dataflow Tasks Aggregation* that was successfully prototyped with our approach for evaluation. The splitting case consists of a RAMSES refinement rule named *Local Communications*. We detail our implementation of this rule to illustrate the splitting case. We first introduce the AADL architecture refinement rule and the refinements of some RDAL requirements for different NFPs and different decompositions into sub-requirements. We then describe the implementation of our co-refinement scheme in terms of a set of orchestrated SDs for the different steps of our co-refinement scheme introduced in section II.C.

1) Architecture Refinement

The *Local Communications* rule refines a message queue communication mechanism (event data port connection in AADL) into a shared data access mechanism prior to code generation for ARINC653 compliant operating systems. Example source and target models are shown in the left part of Fig. 10, where the port connection between two threads executed by the same processor (and therefore communicating locally) is replaced by a data subcomponent shared by the sender and receiver threads via two data access connections. Therefore, the original port connection is split into 3 elements. Additionally, subprogram calls are also added to each thread to implement the refined communication mechanism (not shown on the figure).

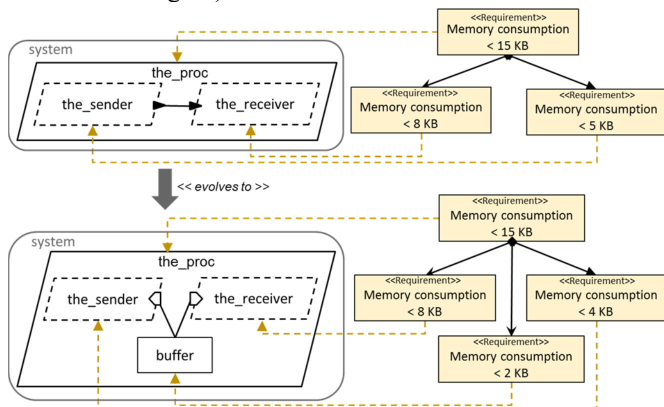

Fig. 10: Initial and refined AADL models for the *Local Communications* rule and assigned memory consumption requirements

C. Considered Requirements

Two types of requirements for the memory consumption and latency NFPs were considered for this architecture refinement rule.

a) Memory Consumption Requirements

For memory consumption, two different requirements structures are considered. In the first case (Fig. 10), a parent requirement is assigned to the process containing the two threads. The requirement is decomposed into 2 sub-requirements for the memory consumption of each thread. Since a data subcomponent (*buffer*) is added to the process during refinement, the total memory consumption of the containing process is increased.

In this case, one possible refinement pattern consists of creating an additional requirement for the buffer. However such refinement cannot be completely automated because the maximum memory allowed for the refined threads, that now have different behaviors through the added subprogram calls and for the added data subcomponent cannot be determined automatically.

For the second case (Fig. 11), we consider a single requirement assigned to both threads. The particularity of this case is that a single requirement is assigned to several architecture elements. One conservative way of refining such requirements consists of simply adding an assignment link to the created buffer data subcomponent and to review its bound value to make sure the original intent of the requirement is preserved. In this case, the bound on memory consumption of each process subcomponent is reduced to 5 KB to ensure the sum of memory consumption does not exceed the initial sum for the two threads.

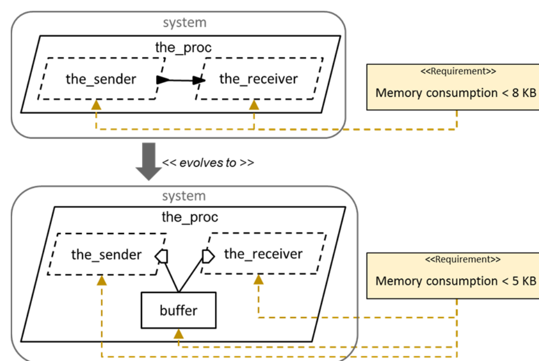

Fig. 11: Memory consumption requirements refinement where the structure of the assignment is updated

a) Latency Requirements

For the latency NFP, an example requirement is assigned to the connection between the threads as shown in the upper part of Fig. 12. A first refinement that can be fully automated consists of reallocating the original requirement to the newly produced buffer subcomponent (lower part of Fig. 12). The OCL constraint expression is also automatically changed to a predefined complex expression that finds the two access connections to the buffer, then computes the sum of their latencies and finally compares the result with the maximum allowed value retrieved from the parsed original OCL expression.

Another refinement pattern could consist of reassigning the requirement to both data access connections and divide the maximum allowed latency value from the original requirement by two.

2) Co-Refinement Scheme Implementation

The implementation of our co-refinement scheme consists of a set of SDs called in the appropriate sequence by a root SD as displayed in Fig. 13. Each node of the SD represents a call to an external SD stored in its own file that implements the corresponding step of the scheme.

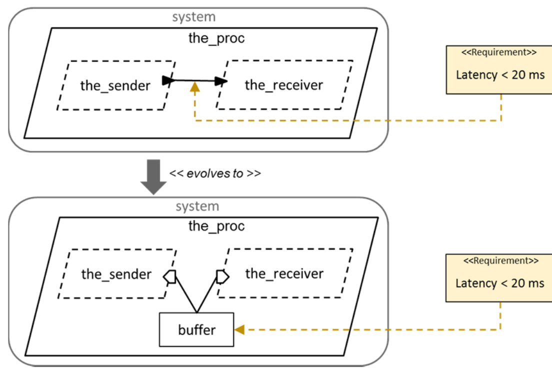

Fig. 12: Example initial and refined latency requirement

The first SD takes care of loading the required AADL library classifiers and property definitions such as the subprogram classifiers implementing the refined communication behavior for the *Local Communications* rule. The required library elements are identified from their names.

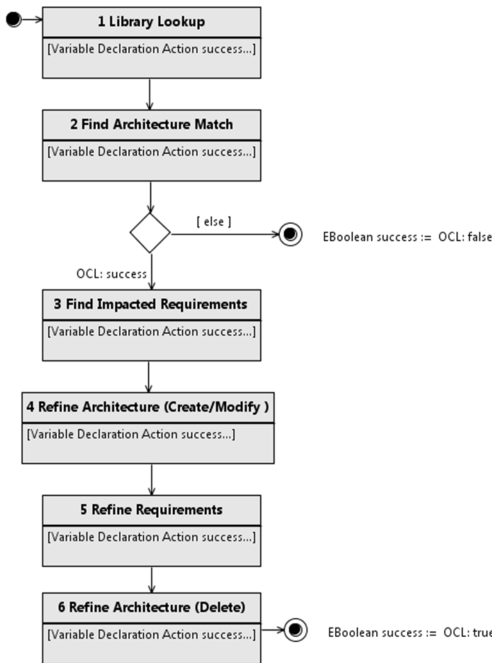

Fig. 13: Overall story diagram implementing the co-refinement scheme

An excerpt of the SD for step 2 of the scheme for finding a match for the architecture refinement rule and for creating the associated markers is illustrated in Fig. 14. The first action node (*match_application_condition*) attempts to match the two communicating threads. An OCL constraint is evaluated on the matched pattern to verify that the matched connection is local, meaning that the connected threads are executed by the same processor.

Once the match is found, the next action node creates the architecture marker model by marking all elements that will be modified during refinement and that will be used to find the potentially impacted requirements during step 3. Such elements are the two communicating threads, the port connection between them and the containing process.

An excerpt of the used architecture marking metamodel is depicted in Fig. 15. It consists of a core metamodel agnostic of any architecture refinement rule and extended by another metamodel specific to the rule. The core metamodel defines an abstract architecture marking class that is a root container of architecture markers. An architecture marker contains a set of modified, deleted, created and matched architecture element references that each refers to a set of architecture elements before refinement (source) and a set of elements produced after refinement (refined). Such marker therefore implements the refinement traceability specified by the co-refinement scheme.

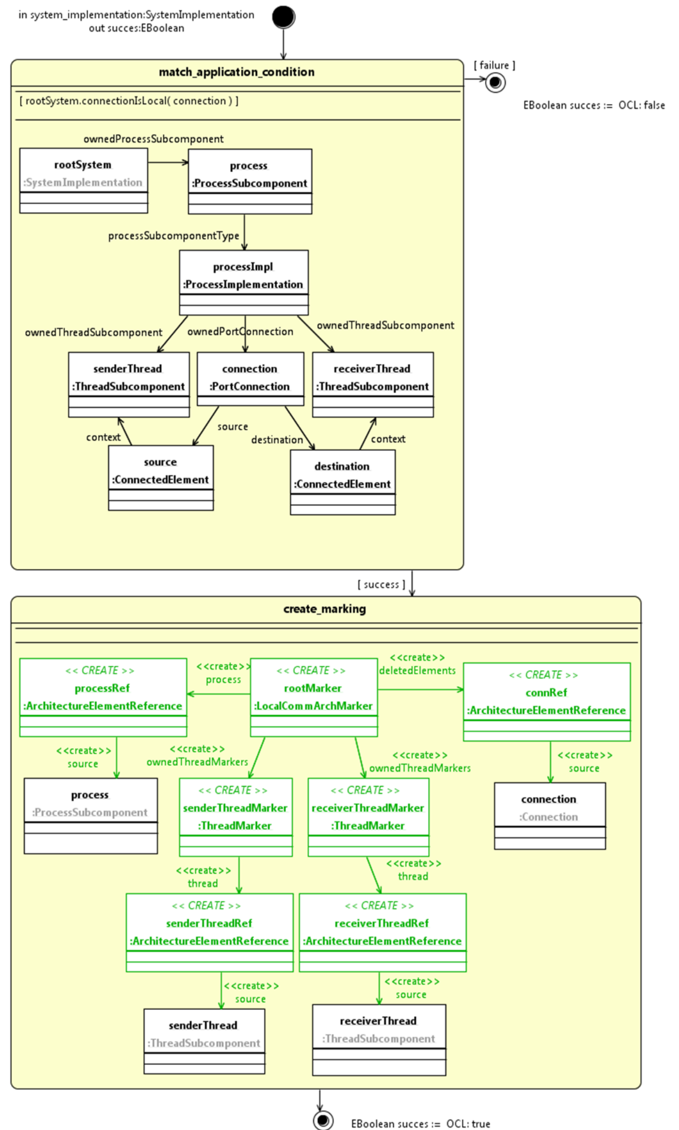

Fig. 14: Simplified version of the story diagram implementing step 2 of the co-refinement scheme for the *Local Communications* refinement rule

For each architecture refinement rule, a concrete architecture marker class is provided extending the core architecture marker class and specific to the architecture refinement rule. Marker classes and references specific to the pattern defining the application condition of the architecture refinement rule are provided as depicted in Fig. 16.

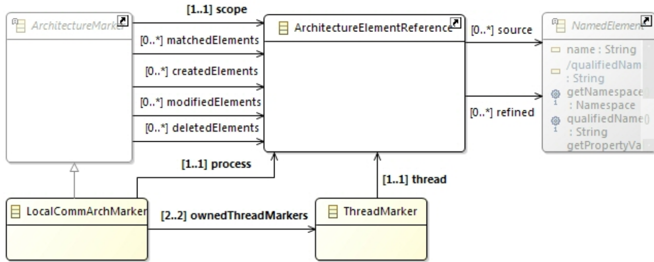

Fig. 15: Excerpt of the architecture marking metamodel for the *Local Communications* refinement rule

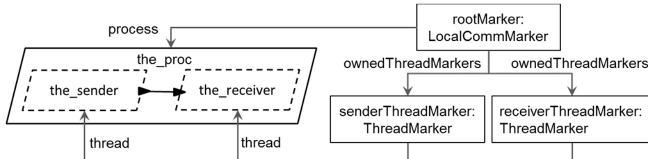

Fig. 16: Marking model for the *Local Communications* refinement rule

Step 3 of the scheme for the detection of impacted requirements is implemented by another SD making use of the information captured by the architecture markers. An excerpt of the requirements marking metamodel is depicted in Fig. 17. Requirements markers are instantiated and identify the impacted requirements for the two communicating threads as well as the latency requirement of Fig. 12. The requirement marker also has a reference to the elements that should be reviewed within the impacted requirement. Various attributes of the requirements marker class are used to store information required by this step of the co-refinement scheme. The architecture elements that will be modified and that are impacting the requirement are also referenced from the requirement marker.

Note that the requirements marking metamodel only makes use of the generic core architecture metamodel and can therefore be reused for any architecture refinement rule.

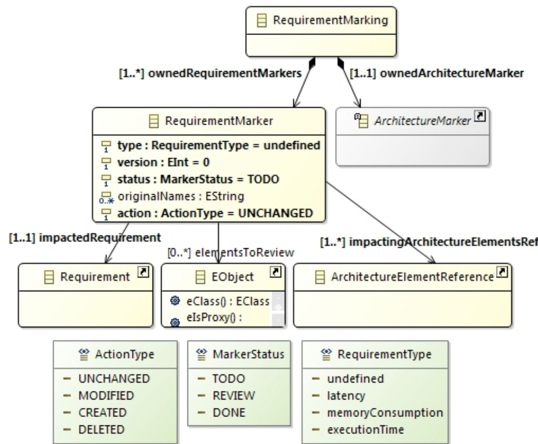

Fig. 17: Excerpt of the requirements marking metamodel

After creating the requirement markers, the architecture refinement rule is executed except for its deleting parts as described in step 4 of the co-refinement scheme. The architecture marking model created during step 2 is used as input. New AADL component type and implementation classifiers are created as copies of the process type and implementation. The required data subcomponent for the buffer is created and added to the new process implementation containing the threads. Its type is set to a library data type previously looked up during step 1. The data access connections are also created and the architecture marker is updated to mark the newly created data subcomponent data access connections and process type and implementation classifiers.

To refine the requirements thus implementing step 5 of the co-refinement scheme, another story diagram is called as shown in Fig. 18. For each requirement marker, the activity calls external story diagrams sequentially until one diagram was found to be able to refine the requirement. Due to the lack of space, we do not show the specific rules implemented for refining the memory consumption and latency requirements.

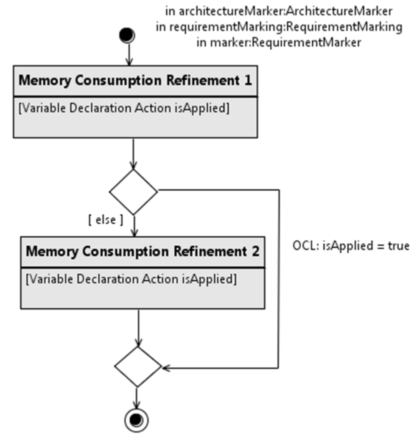

Fig. 18: Excerpt of the SD for refining memory consumption requirements

Lastly, step 6 is applied and the obsolete port connection between the threads is deleted. The delete architecture markers are also updated to replace the references to the deleted elements by their ids.

IV. GENERALIZING REQUIREMENTS REFINEMENT

We have shown in the previous section that our approach can be implemented with SDs and save considerable efforts to the designer by automating partially or not requirements refinements. However, one disadvantage is that for a given architecture refinement rule, many requirements co-refinement rules must be developed specific to the architecture refinement and to the type and structure of the requirements. In this section, we suggest some preliminary ideas for generalizing requirements refinements in order to favor reuse of requirements refinement rules across several architecture refinement rules.

Given the different architecture and requirements refinements that were considered for this work, we foresee two different axis of classification according to the type of architecture refinement (merge vs split of model elements) and to the type of requirements as identified from the constrained NFP. On the type of architecture refinement axis, we noticed that the merge of architecture model elements often leads to merge of requirements and conversely for the splitting case. Such type of applied architecture refinement could therefore

inspire the restructuration of corresponding requirements. On the requirement type axis, NFPs sharing similar characteristics could be identified. Consider for example the period requirement over the merged threads for the merging architecture refinement rule (Fig. 9). When refining the requirements, a single requirement is created for the merged thread with the constraint on the period taken as that of the most restrictive of the source requirements (see section II.A). It is a pattern that could be applied to several types of resource consumption requirements as illustrated in Fig. 19.

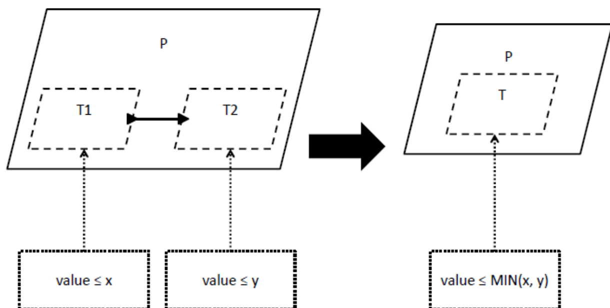

Fig. 19: Pattern for refining requirements following a merge of model elements

Similarly, for the *Local Communications* architecture refinement rule and the considered latency requirement (see section III.C.a)), the initial requirement over the original port connection needs to be split into two new requirements over the two created data access connections, with the sum of the latencies of each connection not exceeding the original value divided by 2. Again this is a pattern applicable for several types of NFPs as illustrated in Fig. 20.

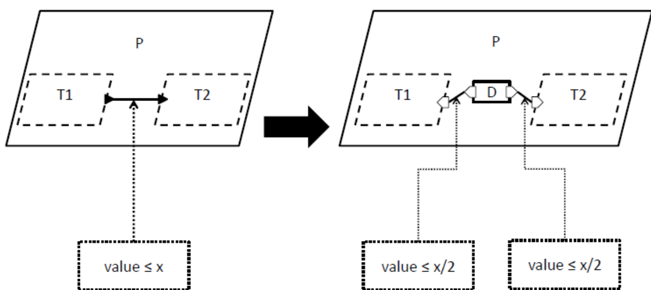

Fig. 20: Pattern for refining requirements following a split of model elements

As another example, consider the reliability of the connection expressed as a number between 0.0 and 1.0, indicating respectively the certainty and impossibility of failure. In such case, splitting the original reliability value would require taking its square root assuming equal reliability for the new connections.

Recalling the refinement of the memory consumption requirement for the *Local Communications* refinement rule (see section III.C.a), if the memory consumption of the added buffer would be known in advance, the maximum memory consumption for the threads after refinement could be computed by subtracting the required memory for the buffer from the initial allowed consumption and divide the result by two for the allowed maximum memory of the refined threads.

V. Related Work

This work relates to the co-evolution of requirements and architectures with a focus on the specific case of refining NF requirements following architecture refinements. We are not aware of any work that addresses this very specific problem. However, there are several works studying the specific nature of the relationship between requirements and architectures [13] and the more general problem of model co-evolution. In the following, we first present the works specific to the co-evolution of requirements and architectures followed by those addressing the general problem of co-evolution of any type of models.

A. Co-evolution of Requirements and Architecture

In [14], an interesting traceability meta-model taking into account the characterization of requirements and architecture elements in terms of problem and solution spaces and capturing design outcomes and decisions is proposed. An ontology supporting the designer in co-evolution is provided. However, only traceability is managed automatically and requirements and architecture must be coevolved manually. In [15], patterns of co-evolution between requirements and source code are proposed. Such patterns provide building blocks for automating traceability maintenance but again, co-evolution of requirements and architecture is not addressed. In [16], a co-evolution of use cases models and feature model configurations is proposed and implemented with a bidirectional transformation language. However this does not address our specific problem and it is not clear how much of the problem is solved by the approach and what level of automation is achieved.

B. Co-evolution for Generic Models

Co-evolution of models is also called model synchronization and a number of approaches have been developed for it. Among those, Triple Graph Grammars (TGG) [17] provides a means to specify declarative rules for transforming one model into another one in both directions. Such rules can be used for co-evolution as they allow relating a change in one model under some conditions to another change in the associated model as required for preserving consistency. TGGs have been used in several co-evolution scenarios such as SysML and AUTOSAR [18]. However TGGs are not expressive enough when the models are of very different domains.

Another approach is EMF DiffMerge / Co-evolution [19]. Similar to TGGs, the approach assumes that the relation between the models can be formalized as an explicit mapping, and synchronization consists of updating the target model according to a source model and the mapping between them. However at the time of writing, no scientific publication could be found for this work that seems to be in a preliminary development stage.

Other approaches to model synchronization are based on defining constraints between models that when evaluated indicate model inconsistencies [20][21][22]. Inconsistency resolution is then performed by requesting the user to provide a repair action to restore consistency or a solver is used to automatically generate such action. However such approaches

are not likely to scale for large models due to the size of the solution space to be explored.

There are also approaches covering only the maintenance of traceability links without considering evolutions of the traced models. In [23], scalable traceability maintenance using Story Diagrams is presented. Tarski [24] goes in a similar direction by enabling the user to specify the semantics of traceability elements using first-order logic.

However, as we have seen in this work, it is rarely the case that repairing traceability is sufficient and requirements must be refined as well, even for the simple cases of architecture refinements we considered. To summarize, no approach propose comprehensive co-evolution of requirements, architecture and their traceability links supporting designers in case complete automation is not possible, including a record of the changes that were applied to the models.

VI. Conclusion and Future Works

We have developed a first approach for co-refining NF requirements following architecture refinements and to support manual refinement by the creation of a marking model when complete automation is not possible. It has been evaluated in the frame of the RAMSES tool with AADL and RDAL models. The approach has been prototyped for several refinement rules indicating its applicability. Furthermore, the approach is highly customizable and is likely to support a large diversity of refinement kinds. It is expected to greatly reduce the manual efforts required for maintaining requirements consistent with the architecture, which is essential for today's large models. It also provides a record of the changes that occurred on both the requirements and architecture sides through the generated marking models.

Our future work consists of improving the approach by increasing its genericity according to the ideas proposed in section IV. For this a thorough study of several architecture refinement rules and requirements is required. The case of functional requirements is also an interesting future work. Such requirements are typically expressed using behavioral constraints languages such as Linear Temporal Logic or Computational Tree Logic, as opposed to structural constraint languages such as OCL for NF requirements. Their refinement could also make use of the proposed co-refinement scheme.

References

[1] A. Kannenberg, H. Saiedian, "Why Software Requirements Traceability Remains a Challenge", CrossTalk: The Journal of Defense Software Engineering, July / August 2009.

[2] Systems Modeling Language (SysML), Object Management Group (OMG), http://www.omgsysml.org/.

[3] A. van Lamsweerde, "Requirements Engineering: From System Goals to UML Models to Software Specifications", Wiley, 2009

[4] E. Borde, S. Rahmoun, F. Cadoret, L. Pautet, F. Singhoff, P. Dissaux, "Architecture models refinement for fine grain timing analysis of embedded systems", Int. Symposium on Rapid System Prototyping (RSP), 2014.

[5] The RAMSES Project Website: https://mem4csd.telecom-paristech.fr/blog/index.php/ramses/,

[6] AS5506b: Architecture Analysis and Design Language (AADL), SAE International, http://standards.sae.org/as5506b/, 2012.

[7] D. Blouin, S. Turki, E. Senn, "Defining an annex language to the Architecture Analysis and Design Language for requirements engineering activities support", Model-Driven Requirements Engineering Workshop (MoDRE), 2011.

[8] T. Fischer, J. Niere, L. Torunski, A. Zündorf, "Story Diagrams A New Graph Rewrite Language Based on the Unified Modeling Language and Java". In Theory and Application of Graph Transformations, 2000.

[9] The Story Driven Modeling Tool (SDM) Project Website, https://www.hpi.uni-potsdam.de/giese/public/mdelab/mdelab-projects/story-diagram-tools/.

[10] D. Blouin, H. Giese, "Combining Requirements, Use Case Maps and AADL Models for Safety-Critical Systems Design", SEAA 2016.

[11] D. Lempia, S. Miller, "Requirements Engineering Management Handbook", Federal Aviation Administration, Tech. Rep., 2009.

[12] The ATL EMFTVM Project Website, https://wiki.eclipse.org/ATL/EMFTVM.

[13] P. Avgeriou, J. Grundy, J. G. Hall, P. Lago. "Relating Software Requirements and Architectures", Springer-Verlag, 2011.

[14] A. Tang, P. Liang, V. Clerc, H. van Vliet, "Traceability in the Co-evolution of Architectural Requirements and Design", in Relating Software Requirements and Architectures, Springer-Verlag, 2011.

[15] M. Rahimi, J. Cleland-Huang, "Patterns of co-evolution between requirements and source code", Procs of the 5th Int. Workshop on Requirements Patterns (RePa), 2015.

[16] W. Zhao, H. Zhao, Z. Hu, "A Framework for Synchronization between Feature Configurations and Use Cases Based on Bidirectional Programming", Procs of the 24th Int. Requirements Engineering Conference Workshops (REW), 2016.

[17] A. Schürr, "Specification of graph translators with triple graph grammars", Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, 1994.

[18] H. Giese, S. Neumann, S. Hildebrandt, "Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent", In Graph Trans. and Model Driven Engineering, Springer, 2010.

[19] The EMF DiffMerge / Co-evolution Project Website, http://wiki.eclipse.org/EMF_DiffMerge/Co-Evolution.

[20] A. Boronat, J. Meseguer, "Automated Model Synchronization: A Case Study on UML with Maude". In Proc. of ECEASST, 2011.

[21] A. Reder, A. Egyed, "Computing repair trees for resolving inconsistencies in design models". In Proc. of the 27th Int. Conference on Automated Software Engineering, 2012.

[22] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, "JTL: A Bidirectional and Change Propagating Transformation Language", In Proc. of the 3rd conference on Software Language Engineering. (SLE), 2010.

[23] A. Seibel, R. Hebig, H. Giese, "Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance", In Software and Systems Traceability, Springer, 2012.

[24] F. Erata, M. Challenger, B. Tekinerdogan, A. Monceaux, E. Tüzün, G. Kardas, "Tarski: A Platform for Automated Analysis of Dynamically Configurable Traceability Semantics", In Proc. of the 32nd Symposium on Applied Computing (SAC), 2017.