Change Impact Analysis for SysML Requirements Models based on Semantics of Trace Relations

David ten Hove[1], Arda Goknil[1], Ivan Kurtev[1], Klaas van den Berg[1] and Koos de Goede[2]

[1]Software Engineering Group, University of Twente, 7500 AE Enschede, the Netherlands
[2]@-portunity B.V., 2023 KB Haarlem, the Netherlands
d.tenhove@student.utwente.nl, {a.goknil, kurtev, k.g.van.den.berg}@ewi.utwente.nl, koos@atportunity.com

Abstract. Change impact analysis is one of the applications of requirements traceability in software engineering community. In this paper, we focus on requirements and requirements relations from traceability perspective. We provide formal definitions of the requirements relations in SysML for change impact analysis. Our approach aims at keeping the model synchronized with what stakeholders want to be modeled, and possibly implemented as well, which we called as the domain. The differences between the domain and model are defined as external inconsistencies. The inconsistencies are propagated for the whole model by using the formalization of relations, and mapped to proposed model changes. We provide tool support which is a plug-in of the commercial visual software modeler BluePrint.

1 Introduction

Requirements traceability is the ability to link requirements back to stakeholders' rationales and forward to corresponding design artifacts, code and test cases [8]. One of the applications of requirements traceability is the change impact analysis. Impact analysis is defined as the process of identifying the potential consequences (side-effects) of a change, and estimating what needs to be modified to accomplish that change [4].

Although considerable research has been devoted to change impact analysis methods using trace relations, less attention has been paid to the usage of trace relation semantics for change impact analysis. In most tools and approaches, there is a lack of precise definition of trace relations. For instance, SysML [12] provides different types of trace relations between requirements, and between requirements & other design artifacts. However, there are only informal definitions for the relations in SysML. In this respect, change impact analysis may result that every related requirement and design artifact are impacted by a requirement change. The cost of implementing a change may become several times higher than expected. Bohner [3] formulates this problem as explosion of impacts without semantics. He states that change impact analysis must employ additional semantic information to increase the accuracy by finding more valid impacts.

In this paper, we focus on requirements and requirements relations from traceability perspective. We give formal definitions of SysML requirements relations [12] [5] in first-order logic. Our approach aims at keeping the model synchronized with what stakeholders want to be modeled, and possibly implemented as well, which we called as the domain. The differences between the domain and model are defined as external inconsistencies. The inconsistencies are propagated for the whole model by using the formalization of relations, and then they are mapped to proposed model changes. The tool support for the approach is a plug-in of the visual software modeler BluePrint [14] developed by @-Portunity.

The paper is structured as follows. Section 2 describes the approach. Section 3 presents the requirements relations in SysML. In Section 4 we provide the formalization for the relations. In Section 5, we describe the use of the formalization for change impact analysis. Section 6 illustrates the approach by an example. In Section 7 we give details of the tool support. Section 8 describes the related work and Section 9 concludes the paper.

2 Overview of the Approach

In this paper, we use the following terminology. ***The Domain*** is what stakeholders want to be modeled, and possibly implemented as well. It is the part of the reality that needs to be modeled, viewed through the requirements it sets for the resulting system. We call the changes in the domain as domain change. ***The Model*** represents a part of the reality called the domain and it is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the domain. ***External Inconsistencies*** define differences between the model and domain. These differences can be caused by a domain change. Our approach is focused on keeping the model synchronized with the domain. ***Internal Inconsistencies*** define conflicts within the model itself. External consistency checking in this paper and internal consistency checking in [6] are complementary.

The Change Impact Analysis process (Figure 1) consists of the following activities:

External Consistency Checking: This activity takes the requirements model and domain change as input, and gives the external inconsistencies as output. The activity has several steps: (1) *identification of a domain change,* which should be performed by a requirements engineer. (2) Then, the requirements engineer *decomposes the domain change into primitive domain change(s)* that we classify as changes to be mapped to proposed model changes. (3) After that, *propagating external inconsistencies* is performed. This step is semi-automatic. Propagation rules are defined based on the formal definitions of the relations. The requirements engineer has to select the correct propagation proposed by the tool.

Model Changing: This activity first handles *mapping the external inconsistencies to the proposed model changes* which are entirely automated. The requirements engineer performs the actual model changes according to proposed model changes.

Iterating: The process given in Figure 1 is iterative. After external consistency checking and model changing, the requirements engineer may return to external consistency checking activity in case there might be new domain changes. Otherwise, the process ends with the assumption that the requirements engineer is satisfied with results.

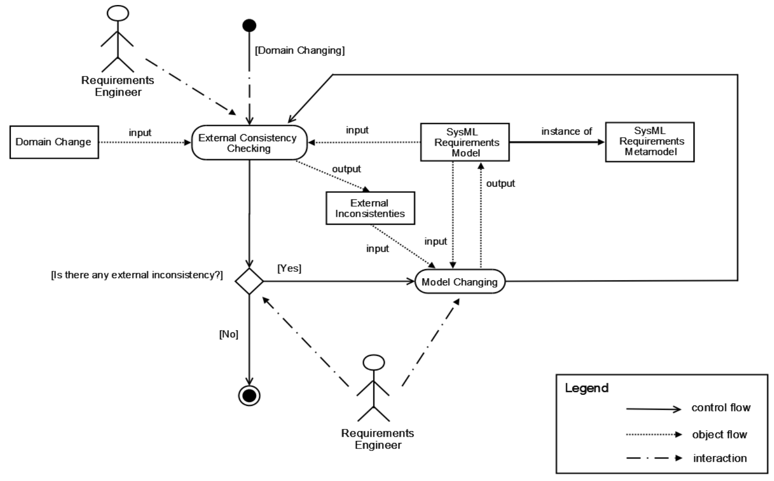


Figure 1 Process for Change Impact Analysis

3 SysML support for Requirements

SysML is a systems modeling language that supports the specification, analysis, design, verification and validation of complex systems. The language is an adaptation of UML for systems including hardware, software, information, and process. In SysML, a requirement is considered as a property that must (or should) be satisfied. The SysML requirements diagram helps in organizing requirements, and also shows explicitly the types of relations between requirements [5]. Figure 2 gives the part of SysML metamodel that depicts the trace relations.

The *Trace* relation provides a general purpose relation between a requirement and any other model element. It has no real constraints and no defined semantics. It is extended by other relations. The relations between requirements in SysML are *ComposedBy*, *Copy* and *DeriveReqt*. Since the *ComposedBy* relation is defined by using UML4SysML::NestedClassifier in SysML metamodel, it is not given as an extension of the *Trace* relation in Figure 2.

These relations are defined in the SysML specification as follows.

- A *ComposedBy* relation enables a complex requirement to be decomposed into its containing child requirements.
- A *DeriveReqt* relation is a dependency between two requirements in which a client requirement can be derived from the supplier requirement.
- A *Copy* relation is a dependency between a supplier requirement and a client requirement that specifies that the text of the client requirement is a read-only copy of the text of the supplier requirement.

These definitions are informal. We formalize the relations in the next section.

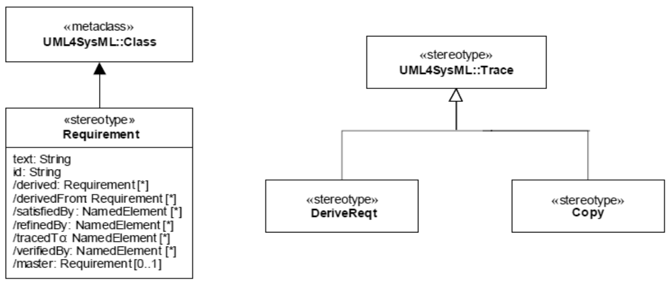

«metaclass»
UML4SysML::Class
«stereotype»
Requirement
text: String
id: String
/derived: Requirement [*]
/derivedFrom: Requirement [*]
/satisfiedBy: NamedElement [*]
/refinedBy: NamedElement [*]
/tracedTo: NamedElement [*]
/verifiedBy: NamedElement [*]
/master: Requirement [0..1]
«stereotype»
UML4SysML::Trace
«stereotype»
DeriveReqt
«stereotype»
Copy

Figure 2 Part of SysML Metamodel for Requirements Diagrams [12]

4 Formalization of Requirements and Relations

In this section we provide the formalization for requirements and the *DeriveReqt* relation. The other relations are formalized in a similar way (but they are not presented here due to space limitation).

We chose a formalization of requirements in first-order logic (FOL). The expressiveness of FOL is sufficient for our goal. There are examples of formalization of requirements in other types of logic such as modal and deontic logic [11].

We assume the general notion of a requirement in SysML being "a property that must (or should) be satisfied". We define a requirement R as a tuple <P, S> where P is the property and S is the set of systems that satisfy P, i.e. $\forall s \in S : P(s)$. P can be represented in a conjunctive normal form (CNF) in the following way:

$$P = (p_1 \land \dots \land p_n);\text{ where } n \geq 1 \text{ and } p_n \text{ is disjunction of literals}$$

A literal is an atomic formula (atom) or its negation. An atomic formula is a predicate symbol applied over terms. We assume that all formulas are in CNF. In the rest of the paper we use the notation $(p_1 \dots p_n)$ for $(p_1 \land \dots \land p_n)$.

We formalize the *DeriveReqt* relation as follows: Let $R_1 = <P_1, S_1>$ and $R_2 = <P_2, S_2>$ be requirements. P_1 and P_2 are formulas and the conjunctive normal form of P_2 is:

$$P_2 = (p_1..p_n) \land (q_1..\, q_m);\ \ n \geq 1, m \geq 0$$

Let $p_1^I, p_2^I, \dots, p_{n-1}^I, p_n^I$ be disjunction of literals such that $p_j^I \rightarrow p_j$ for $j \in 1..n$

R_1 *DeriveReqt* R_2 iff P_1 is derived from P_2 by replacing every p_j in P_2 with p_j^I for $j \in 1..n$ such that the following two statements hold:

a) $P_1 = (p_1^I..\, p_n^I) \land (q_1..\, q_m) \land (z_1..\, z_t);\ \ n \geq 1, m \geq 0, t \geq 0$

b) $\exists s \in S_2 : s \notin S_1$

From the definition we conclude that if P_1 holds for a given system *s* then P_2 also holds for *s* ($\forall s \in S_1 : s \in S_2$). On the basis of $\exists s \in S_2 : s \notin S_1$ and $\forall s \in S_1 : s \in S_2$, we conclude ($S_1 \subset S_2$). We have the properties *non-reflexive*, *non-symmetric*, *transitive* for the *DeriveReqt* relation.

5 Change Impact Analysis

Change impact analysis is defined as the process of identifying the potential consequences (side-effects) of a change, and estimating what needs to be modified to accomplish that change [4]. Analyzing the impact of changes provides determining possible conflicts and design alternatives influenced by changes. Our change impact analysis approach is based on determining external inconsistencies for a domain change and proposing possible model changes to fix these inconsistencies between the domain and model. In this respect, propagating external inconsistencies based on the semantics of relations are the potential consequences of a change and proposed model changes are the estimates about what needs to be modified to accomplish the change. Table 1 shows how domain changes are mapped to external inconsistencies, and in turn how external inconsistencies are mapped to model changes.

Table 1 Domain Changes, External Inconsistencies and Model Changes

#	Primitive Domain Change	External Inconsistency	Model Change
1	New requirement added to the domain	Requirement in the domain but absent in the model	Requirement is added
2	Existing requirement is removed from the domain	Requirement is not in the domain but present in the model	Requirement is removed
3	Requirement in the domain is made more specific	Requirement in the model is less specific than the requirement in the domain	Details are added to the requirement
4	Requirement in the domain is made more abstract	Requirement in the model is more specific than the requirement in the domain	Details are removed from the requirement
5	Part is removed from the requirement in the domain	Requirement in the model has more parts than the requirement in the domain	Part is removed from the requirement
6	New part is added to the requirement in the domain	Requirement in the model has less parts than the requirement in the domain	Part is added to the requirement

The domain changes in Table 1 are called primitive domain changes. Mapping a domain change to an external inconsistency according to Table 1 is manually done by the requirements engineer. First, the requirements engineer determines the domain change and decomposes the domain change into primitive domain changes. With the help of Table 1, he/she determines external inconsistencies between the domain and model. Mapping an external inconsistency to a model change is done automatically. However, there might be other external inconsistencies derived from the external inconsistency identified by the requirements engineer.

In Section 4, we give formal definition of SysML requirements relations in first-order logic. Based on the formal definition of the relations, we define external inconsistency propagation rules. Table 2 shows the external inconsistency propagation rules for the “DeriveReqt” relation. Similar tables are derived for other relations (but they are not presented here due to space limitation). The columns in Table 2 are the external inconsistency propagation rules for external inconsistency types.

Table 2 External Inconsistency Propagation Rules for the “DeriveReqt” Relation

#	External Inconsistency	R_1 DeriveReqt R_2	R_1 DeriveReqt R_2 … R_n
1	R_1 is not in the domain	R_2 is not in the domain	R_2 … R_n are not in the domain
2	R_2 is not in the domain	R_1 is not in the domain or part of R_1 is not in the domain	Part of R_1 is not in the domain or R_1 is not in the domain
3	R_1 is less specific than it is in the domain	No propagation	No propagation
4	R_2 is less specific than it is in the domain	R_1 is less specific than it is in the domain	R_1 is less specific than it is in the domain
5	R_1 is more specific than it is in the domain	R_2 is more specific than it is in the domain or no propagation	(R_2 is more specific than it is in the domain and/or R_n more specific than it is in the domain) or no propagation
6	R_2 is more specific than it is in the domain	R_1 is more specific than it is in the domain	R_1 is more specific than it is in the domain
7	R_1 has more parts than it has in the domain	R_2 has more parts than it has in the domain or R_2 is not in the domain or no propagation	((R_2 not in domain or part of R_2 not in domain) and/or (R_n not in domain or part of R_n not in domain)) or no propagation
8	R_2 has more parts than it has in the domain	R_1 has more parts than it has in the domain	R_1 has more parts than it has in the domain
9	R_1 has less parts than it has in the domain	No propagation	No propagation
10	R_2 has less parts than it has in the domain	R_1 has less parts than it has in the domain	R_1 has less parts than it has in the domain or no propagation
11	Relation is in the model, not in the domain	No propagation	No propagation
12	R_4 is in the domain, not in the model	No propagation	No propagation

Some external inconsistencies like in Rule 9, Rule 11 and Rule 12 do not propagate while others like in Rule 2, Rule 5, Rule 7 and Rule 10 have multiple propagation possibilities. All these rules are defined based on the semantics of the “DeriveReqt” relation given in first-order logic. Due to space limitation we can not give explanation of the propagation rules in Table 2. The following explains how Rule 4 in Table 2 for the “*R_1 DeriveReqt R_2*” case is defined.

Let $R_1 = <P_1, S_1>$ and $R_2 = <P_2, S_2>$ be requirements. Since *R_1 DeriveReqt R_2*, we have P_1 and P_2 in the following conjunctive normal form.

$P_2 = (p_1..p_n) \wedge (q_1.. q_m)$; $n \geq 1, m \geq 0$
$P_1 = (p_1^I.. p_n^I) \wedge (q_1.. q_m) \wedge (z_1.. z_t)$; $n \geq 1, m \geq 0, t \geq 0$
where $p_1^I, p_2^I, \ldots, p_{n-1}^I, p_n^I$ be disjunction of literals such that $p_j^I \rightarrow p_j$ for $j \in 1..n$

Rule 4 in Table 2 has the external inconsistency “R_2 is less specific in the model than it is in the domain”. According to Table 1, this external inconsistency is caused by the domain change “Requirement in the domain is made more specific”.

After the domain change, at least one of the disjunctions of literals in the conjunction normal form of P_2 (p_n or q_m) is less specific than it is in the domain.

Since we have $P_1 = (p_1^I.. p_n^I) \wedge (q_1.. q_m) \wedge (z_1.. z_t)$, at least one of the disjunctions of literals in the conjunction normal form of P_1 (p_n^I or q_m) is less specific than it is in the domain

This means adding a detail to R_1, by tagging it as “R_1 is less specific than it is in the domain”.

The domain change and external inconsistency, together, provide the reason of the model change. Mapping the external inconsistency to the proposed model changes justifies the model change. Therefore, we choose propagating the external inconsistency rather than propagating the model change. When we know all parts of the model to be changed, we can provide the proposed changes for the whole model.

6 Example

In this section we illustrate our approach by a well-known example using the requirements for a Rain Sensing Wiper (RSW) system [2]. The goal of the RSW system is to wipe the surface of the windshield automatically whenever droplets of liquid are detected on the windshield's surface. The amount of liquid detected dictates the speed of the wiper. Balmelli [2] gives the example requirements model in SysML for the Rain Sensing Wiper system. The textual form of the requirements for the Rain Sensing Wiper system can be found in Appendix 1.

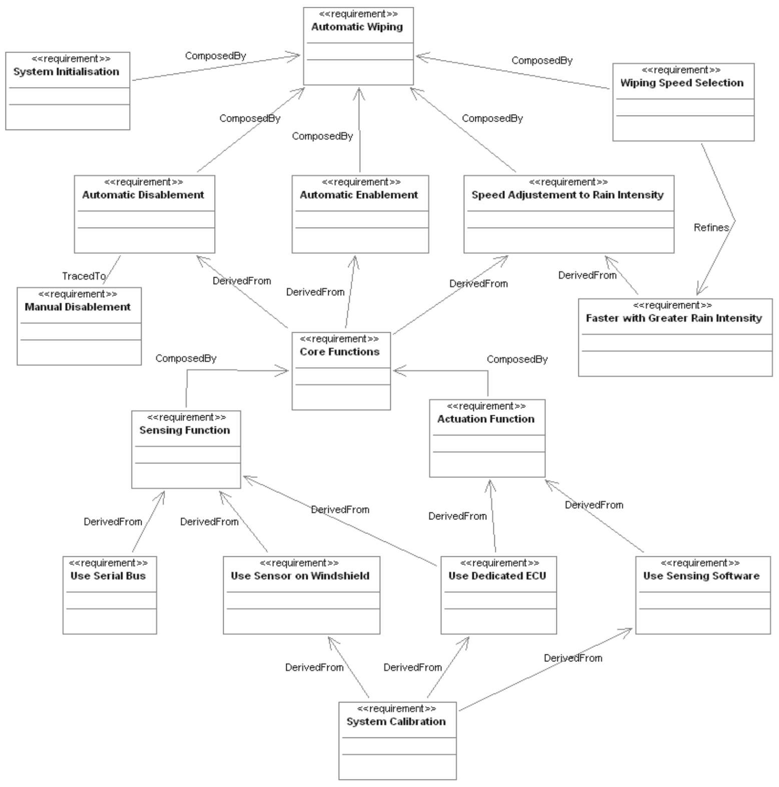

<<requirement>> Automatic Wiping
<<requirement>> System Initialisation
<<requirement>> Wiping Speed Selection
<<requirement>> Automatic Disablement
<<requirement>> Automatic Enablement
<<requirement>> Speed Adjustement to Rain Intensity
<<requirement>> Manual Disablement
<<requirement>> Faster with Greater Rain Intensity
<<requirement>> Core Functions
<<requirement>> Sensing Function
<<requirement>> Actuation Function
<<requirement>> Use Serial Bus
<<requirement>> Use Sensor on Windshield
<<requirement>> Use Dedicated ECU
<<requirement>> Use Sensing Software
<<requirement>> System Calibration
ComposedBy
DerivedFrom
TracedTo
Refines

Figure 3 SysML Requirements Model for the Rain Sensing Wiper System [2]

We give one change scenario to illustrate our approach. For external inconsistency propagation, we give only the explanation of the *DeriveReqt* relation in the example.

Activity 1: External Consistency Checking:

Step 1: Identify a domain change

- We have the domain change for the actuator functions "*C-language will be used for actuation functions (determined by user)*"

Step 2: Decomposing the domain change into primitive domain changes

- The domain change itself is a primitive domain change. We classify this domain change as "*Requirement in the domain is made more specific*"
- The external inconsistency is applied to the requirement in the model "*Actuation function*"
- The domain change is mapped to the external inconsistency "*Requirement in the model is less specific than it is in the domain*" (automatically derived from Rule 3 in Table 1)

Step 3: Propagate external inconsistencies

- Same external inconsistency is propagated from **Actuation function** to **Use Dedicated ECU** (automatically derived from Rule 4 in Table 2)
- Same external inconsistency is propagated from **Actuation function** to **Use Sensing software** (automatically derived from Rule 4 in Table 2)
- Same external inconsistency is propagated from **Use Sensing Software** to **System Calibration** (automatically derived from Rule 4 in Table 2)
- Same external inconsistency is propagated from **Use Dedicated ECU** to **System Calibration** (automatically derived from Rule 4 in Table 2). This requirement was already tagged as externally inconsistent in the same way
- No propagation from **Use Dedicated ECU** to **Sensing Function** (automatically derived from Rule 3 in Table 2)
- No propagation from **System Calibration** to **Use Sensor on Windshield** (automatically derived from Rule 3 in Table 2)
- No propagation from **Core Functions** to **Automatic Disablement** and **Automatic Enablement** and **Speed Adjustment to Rain Intensity** (automatically derived from Rule 3 in Table 2)

Activity 2: Model Changing

Step 1: Map external inconsistencies to model changes

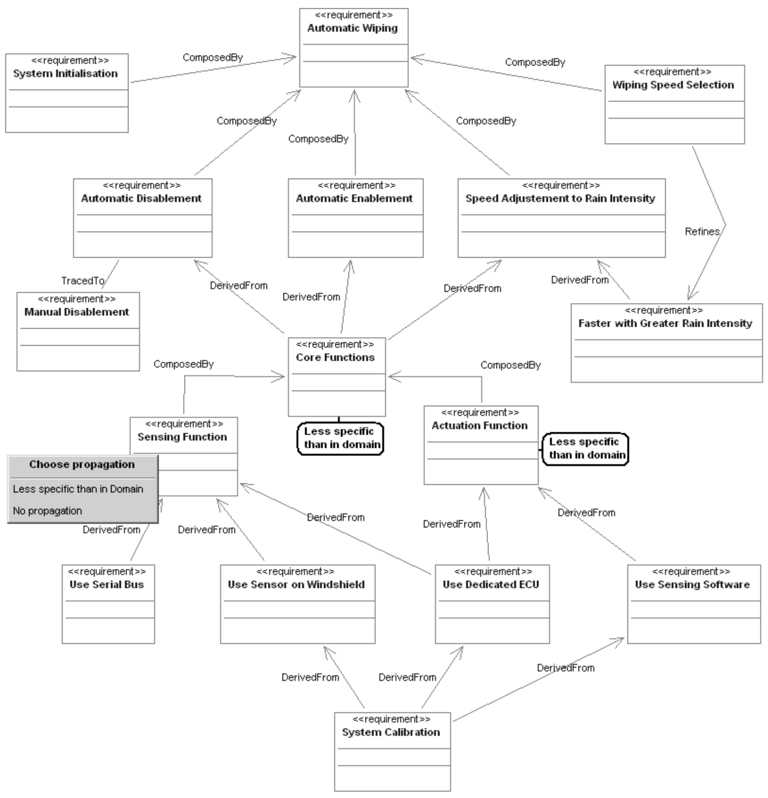
- We have only one external inconsistency type: "*Requirement in the model is less specific than it is in the domain*". Each one is mapped to the model change "Details are added to the requirement" (automatically derived from Rule 3 in Table 1)

Step 2: Implementing the model changes

- The only assistance, here, is the type of the model change which should be performed. Apart from that, the implementation of model changes is manual.

7 Tool support

We provide the tool support for change impact analysis in SysML requirements models. In this section, we depict the usage of the tool within the context of the process given in Figure 1. The tool support is a plug-in of the UML2.1 compliant visual software modeler BluePrint [14]. The tool supports the *propagating external inconsistency* step in the *external inconsistency checking activity* and the *mapping external inconsistencies to proposed model changes* step in the *model changing* activity. Figure 4 gives the output of the external inconsistency propagation for the RSW system example.



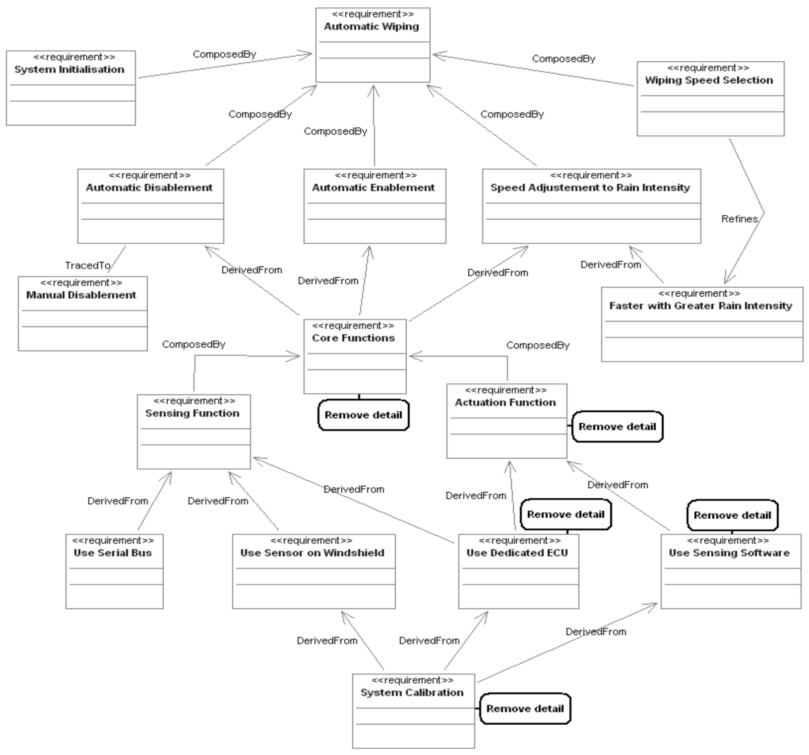
<<requirement>> Automatic Wiping
<<requirement>> System Initialisation
<<requirement>> Wiping Speed Selection
ComposedBy
ComposedBy
ComposedBy
ComposedBy
ComposedBy
<<requirement>> Automatic Disablement
<<requirement>> Automatic Enablement
<<requirement>> Speed Adjustement to Rain Intensity
Refines
TracedTo
<<requirement>> Manual Disablement
DerivedFrom
DerivedFrom
DerivedFrom
DerivedFrom
<<requirement>> Faster with Greater Rain Intensity
<<requirement>> Core Functions
ComposedBy
ComposedBy
<<requirement>> Sensing Function
Less specific than in domain
<<requirement>> Actuation Function
Less specific than in domain
Choose propagation
Less specific than in Domain
No propagation
DerivedFrom
DerivedFrom
DerivedFrom
DerivedFrom
DerivedFrom
<<requirement>> Use Serial Bus
<<requirement>> Use Sensor on Windshield
<<requirement>> Use Dedicated ECU
<<requirement>> Use Sensing Software
DerivedFrom
DerivedFrom
DerivedFrom
<<requirement>> System Calibration

Figure 4 Output of the External Inconsistency Propagation

The external inconsistency “Less Specific than in Domain” for the Actuation Function requirement is propagated into the Core Functions requirement as “Less Specific than in Domain”. The rounded boxes give the external inconsistencies. In Figure 4, the popup window lists the alternative propagations for the *Sensing Function* requirement. The requirements engineer selects the appropriate one.

After determining all external inconsistencies in the model, the tool derives the proposed model changes from these inconsistencies based on the mapping given in Table 1. Figure 5 gives the output of the proposed model changes. The rounded boxes tag the *Core Functions*, *Actuation Function*, *Use Sensing Software*, *Use Dedicated ECU* and *System Calibration* requirements with the proposed model change “*Remove*

Detail". The requirements engineer does the actual changes with the help of the proposed model changes.



<<requirement>> Automatic Wiping
<<requirement>> System Initialisation
<<requirement>> Wiping Speed Selection
<<requirement>> Automatic Disablement
<<requirement>> Automatic Enablement
<<requirement>> Speed Adjustement to Rain Intensity
<<requirement>> Manual Disablement
<<requirement>> Faster with Greater Rain Intensity
<<requirement>> Core Functions
<<requirement>> Sensing Function
<<requirement>> Actuation Function
<<requirement>> Use Serial Bus
<<requirement>> Use Sensor on Windshield
<<requirement>> Use Dedicated ECU
<<requirement>> Use Sensing Software
<<requirement>> System Calibration
ComposedBy
Refines
TracedTo
DerivedFrom
Remove detail

Figure 5 Output for the Proposed Model Changes

8 Related Work

In our previous work [6], we proposed a metamodel for requirements models (called *core* metamodel). We define the semantics of the concepts and the relations in the core metamodel. On the basis of the semantics we can perform reasoning on requirements that may detect implicit relations and internal inconsistencies. However, the approach in [6] does not support change impact analysis. As a continuum of that work, we proposed a change impact analysis technique [7] based on formalization of requirements relations in the requirements metamodel in [6]. However, the change impact analysis technique in [7] aims at propagating the impact of the change directly rather than propagating the external inconsistency. We did not have the concepts like domain, external inconsistency that provide the reason of the change. Therefore, without the reason of the change in [7], the approach gives similar impacts for different types of changes.

Ajila [1] explicitly defines elements and their relations to be traced with dependencies they called as intra-level and inter-level. Impact analysis based on transitive closures of call graphs is discussed in Law [9]. Lindvall et al. [10] show tracing across phases again with intra-level and inter-level dependencies. They also discuss an impact analysis method based on traceability. However, they do not support their analysis with formalism. Change impact analysis for software architectures has been studied by Zhao et al. [13]. They use a formal architectural description language to specify the architectures.

9 Conclusion

In this paper, we proposed a change impact analysis technique based on formalization of requirements relations considered as trace relations in SysML. The approach focuses on requirements models reflecting the domain what stakeholders want to be modeled, and possibly implemented as well. Any model changes are fueled by changes in that domain.

Using the formal definitions of the SysML relations *ComposedBy*, *Copy* and *DeriveReqt*, several change impact rules were defined. These rules give the propagation possibilities of external inconsistencies which define differences between the model and domain. They are mapped to model changes. The requirements engineer is guided through the change process using these rules. He only needs to select the proper propagation rules. Implementing model changes puts the model back in sync with the domain.

Since the approach is based on SysML, existing tools can be easily extended in order to include it. This was shown in the tool support. The tool support is a plug-in of the visual software modeler BluePrint [14]. We applied our approach to an example SysML requirements model for the Rain Sensing Wiper system.

References

1. Ajila, S.: Software Maintenance: An Approach to Impact Analysis of Object Change. Software - Practice and Experience, 25(10), pp. 1155-1181, 1995
2. Balmelli, L.: An Overview of the Systems Modeling Language for Products and Systems Development. Journal of Object Technology, 6(6), pp. 149-177, 2007.
3. Bohner, S.A.: Software Change Impacts – An Evolving Perspective. ICSM’02, IEEE Computer Society Press, pp. 263-271, 2002.
4. Briand, L.C., Labiche, Y., O`Sullivan, L., Sowka, M.M.: Automated Impact Analysis of UML Models. Journal of Systems and Software, 79(3), pp. 339-352, 2006.
5. dos Santos Soares, M., Vrancken, J.: Model-Driven User Requirements Specification using SysML. Journal of Software, 3(6), pp. 57-68, June 2008.
6. Goknil, A., Kurtev, I., van den Berg, K.: A Metamodeling Approach for Reasoning about Requirements. European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA’08), LNCS, vol. 5095, pp. 311-326, 2008.
7. Goknil, A., Kurtev, I., van den Berg, K.: Change Impact Analysis based on Formalizations of Trace Relations for Requirements. ECMDA-TW’08, SINTEF Report, pp.59-75, 2008.

8. Gotel, O.C.Z., Finkelstein, C.W.: An Analysis of the Requirements Traceability Problem. RE'94, IEEE Computer Society Press, pp. 94-101, 1994.
9. Law, J., Rothermel, G.: Whole Program Path-based Dynamic Impact Analysis. ICSE'03, IEEE Computer Society Press, pp. 308-318, 2003
10. Lindvall, M., Sandahl, K.: Traceability Aspects of Impact Analysis in Object-oriented Systems. Software Maintenance: Research and Practice, vol.10, pp. 37-57, 1998.
11. Meyer, J.J.C., Wieringa, R., Dignum, F.: The Role of Deontic Logic in the Specification of Information Systems. Logics for Databases and Information Systems, pp.71-115, 1998.
12. OMG: SysML Specification. OMG ptc/06-05-04, http://www.sysml.org/specs.htm
13. Zhao, J., Yang, H., Xiang, L., Xu, B.: Change Impact Analysis to Support Architectural Evolution. Journal of Software Maintenance and Evolution: Research and Practice, vol.14, pp. 317-333, 2002
14. @-portunity. http://www.atportunity.com/

Appendix 1. Rain Sensing Wiper System Requirements

In this appendix, we give an overview of the textual form of the Rain Sensing Wiper system requirements modeled as SysML requirements model (based on [2]).

R1 Automatic wiping: The system shall automatically wipe the windshield of the car whenever necessary or desired by the user
R2 System Initialization: System initial check-up
R3 Automatic Disablement: The system shall automatically stop wiping the windshield when it is no longer necessary
R4 Manual Disablement: The driver should be able to stop the wiping manually
R5 Automatic Enablement: The system shall automatically start wiping the windshield when it is necessary
R6 Wiping Speed Selection: The system shall offer three different wiping speeds from which the driver can choose
R7 Speed Adjustment to Rain Intensity: The wiping speed should adjust according to rain intensity
R8 Faster with Greater Rain Intensity: The more rain, the faster the wiping
R9 Core functions: Identified core functions
R10 Sensing function: The system shall be able to sense rain intensity
R11 Actuation Function: The system shall be able to actuate based on automatic and manual input
R12 Use Serial Bus: The system shall use a serial bus to transfer data
R13 Use Sensor on Windshield: The system shall sense the rain intensity via a sensor on the windshield
R14 Use Dedicated ECU: An Electronic Control Unit dedicated to this purpose will serve as the processor for the input
R15 Use Sensing Software: A software solution shall be implemented to process driver and sensor input
R16 System Calibration: The sensor shall be calibrated for the characteristics of the windshield, and the type of car