# Extending the NFR Framework with Measurable Non-Functional Requirements

Anton Yrjönen and Janne Merilinna

VTT Technical Research Centre of Finland
P.O.Box 1100
FI-90571 Oulu, Finland
{anton.yrjonen, janne.merilinna}@vtt.fi

**Abstract.** Accurate and correctly specified requirements are extremely important in ensuring the production of feasible software products. To assure that the requirements have actually been implemented, there has to be a trace link from requirements to implementation. Thus far requirement engineering has been a rather separate task from software design and implementation from the process point of view. This separation has a negative impact on requirements traceability and further, to product quality. Tracing of non-functional requirements (NFRs), such as performance, has been particularly cumbersome. Thus, in this paper we apply and extend the NFR Framework to bridge the gap between NFRs and implementation. We have implemented the extended NFR Framework, which we call NFR+ Framework, as a modelling language including a softgoal interdependency graph validation tool with a MetaCase MetaEdit+ language workbench. We extended the NFR Framework with a concept of measurable NFRs that enables to empirically verify the realization of defined NFRs in a product. The usage of the extended NFR Framework is demonstrated with a laboratory case.

**Keywords:** Domain-Specific Modelling, requirements engineering, tracing

## 1 Introduction

Traditionally requirements engineering (RE) and software engineering (SE) are seen quite separate areas of research, profession and action [1] [2]. This has resulted in a separation of the tools, processes and methods used in each of the tasks. For each purpose, individual tools have been created but the information flow and integrity from one task and tool to another has quite often lagged behind tool development and has often been realized afterwards in an inconvenient way, if at all. Due to lack of sufficient tool support and integrated processes, the same work may have been done twice, overlapping and unsynchronized. Designers might have had challenges to figure out what the requirements mean in terms of implementation and what actions are needed to achieve the desired level of satisfaction. The challenge is highlighted when concerning more ambiguous non-functional requirements (NFR) that concern the non-functional, a.k.a. quality attributes of the software [3]. It is often unclear what side-effects there might be for choosing a certain solution to fulfil a requirement or how important and for what reason a NFR really is. In the worst case, there may not be concrete and accurate enough NFRs at all or the designers are not aware of them.

The interconnection of requirements and software design is more than the simplest junction of requirements, that being the output of RE and input for SE. The connection exists another way around in software testing and requirements verification, when the evaluation concerns the question whether the defined requirements really are met by the software [4][5]. Since requirements management should be closely bound to the verification, there is a missing link between requirements and verification, which is also closely related to the software design.

The NFR Framework is a systematic approach for producing and documenting proper NFRs through graphical modelling [6]. However, there is a danger that applying NFR Framework produces requirements which can be difficult to evaluate being realized. Yet, all requirements should be expressed in terms of measurable properties to enable verification [7]. In the case of NFRs these can be applicable numeric measurements of performance, resource consumption, correctness, reliability, security etc. Preferably there should be threshold values for success/failure evaluation together with the measurement arrangements. This promotes the avoidance of interpretation conflicts between stakeholders and promotes instead the definition of accurate, realistic and useful requirements in the place of overly optimistic or vague requirements. A good requirement should be verifiable and unambiguous, quantifiable and measurable. This closes the loop between RE, design and verification, which is necessary to ensure the quality of the product.

Without a common, solid tool environment, the requirements may end up being stored in separate documents that are not commonly referred to by designers and may be outdated after a short while. For designers, the requirements should be transparently traceable to the original rationale as well as to other related design entities.

To overcome these challenges, our common goal is combining RE and SE workspaces and information flow. Since the NFR Framework is essentially about graphical modelling, it is convenient to use a graphical Model-driven Development (MDD) environment to implement the NFR Framework. MDD on the other hand is based on the need to raise the design abstraction level closer to the problem domain. Domain-Specific Modelling in particular (DSM) [8] has a promising capability for adopting the problem relevant language constraints for describing the system under development. Since SE with DSM already use problem terminology (instead of solution terminology, such as, classes and objects), the mapping from requirements to solution seems possibly straightforward. Thus we utilize DSM for SE in this case.

In this paper, not only do we present the implementation of the NFR Framework in MetaCase MetaEdit+[1] language workbench, but we also extend the NFR Framework Softgoal Interdependency Graph (SIG) with a concept of measurable NFRs to form a NFR+ Framework. The measurable NFR provides evidence-based information for a requirements engineer to determine if the defined NFRs are achieved. They also serve as a connection point and guide to software designers by stating the desired outcome in terms of NFRs. It is argued that the introduced link to measurable NFRs is an important step to close the gap between RE and SE. The NFR+ Framework is

[1] http://www.metacase.com

demonstrated in a laboratorial case study of stream-oriented image processing applications.

The rest of the paper is structured as follows. In Section 2 the NFR Framework is described to set the background and baseline for our work. In Section 3, an extension for the NFR Framework is presented including the implementation details. The usage of the extension is demonstrated with a laboratorial case example in Section 4. The applicability of the developed solution is discussed in Section 5 and the paper is concluded in Section 6.

## 2 The NFR Framework

The NFR Framework offers a systematic approach for defining NFRs for products. It offers good visibility to all relevant NFRs and their interdependencies, and helps designers to understand the necessary actions for ensuring proper quality. It also captures and documents design decisions and rationales in addition to providing traceability for derived specifications and requirements. [6]

### 2.1 Softgoal Interdependency Graph

The NFR Framework is mainly based on the SIG which is a graph of interconnected softgoals where each represents an NFR for the system under development. During requirements elicitation, abstract softgoals are decomposed to more detailed and concrete child softgoals which contribute with positive or negative impact or alternatively with logical AND/OR compositions to the parent softgoal.

Each softgoal has a type and a topic. Decomposing a softgoal into subtopical and more specific types of softgoals, leads ultimately to so called operationalizations at the end nodes of the graph. Operationalizations are different kinds of implementation and design techniques that can be selected to ensure a feasible outcome. The operationalizations can be used as specifications for the system.

Figure 1 shows an example of a SIG as it appeared in the original book [6]. The cloud symbols drawn with thin lines are abstract NFR softgoals and the thicker clouds are concrete operationalizations. The relevant topic is annotated within square brackets that follow the type declaration. Within the NFR Framework itself specific catalogues for type and topic decompositions can exist, which state the common, allowed or desired decompositions.

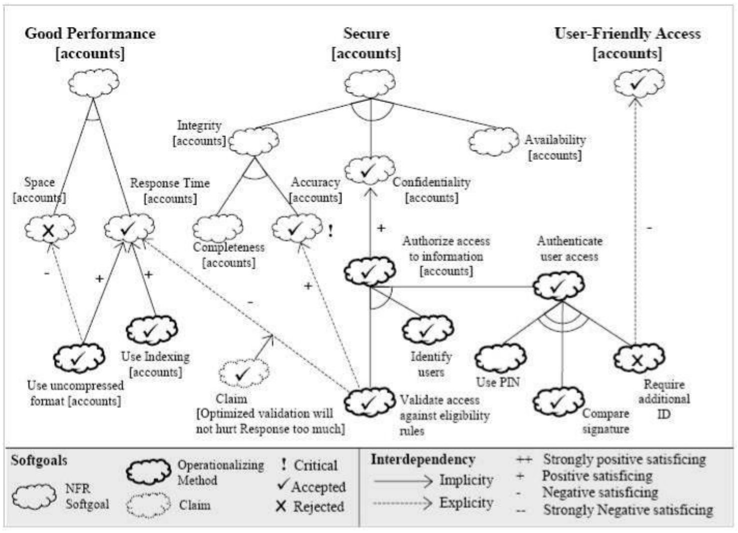


**Figure 1. Softgoal Interdependency Graph with a legend of symbols.**

As the softgoals are decomposed during the NFR elicitation, the decomposition relationships connecting the softgoals and operationalizations state the interdependencies between the softgoals and operationalizations. When there is an arrow or a line, this means that the connected symbols have an interdependency the direction of which is towards the tip of an arrow; from the more concrete and specific, to the more generic and abstract softgoal. Instead of bilateral (connecting only two symbols) interdependency, there can also be logical AND/OR interdependencies that have exactly one higher level softgoal symbol attached to two or more subgoals. In addition to interdependencies between softgoals, Claims can be inserted to justify certain decisions of labels. In figure 1, for example, there is a Claim that the negative contribution of "Validate access" does not harm "Response time" significantly.

Softgoals may be annotated with a label. The labels (see Table 1) are used to inform the status of softgoals and operationalizations. The label typically denotes a decision to select a certain method or technique for operationalizing a softgoal. For the more abstract softgoals, the label describes the status of all the other NFRs logically underneath it, since the children's effect propagates upwards in the SIG through interdependencies.

**Table 1. Softgoal labels**

| Symbol | Name | Explanation |
|---|---|---|
| √ | Satisfied | The softgoal is fulfilled or chosen to be implemented. |
| w+ | Weakly satisfied | There is some positive support to the softgoal. |
| u | Undecided | Realization of the softgoal neither confirmed nor denied. |
| w- | Weakly denied | There are some indicators against fulfilling the softgoal. |
| x | Denied | Softgoal can not be realized or is chosen not to be implemented. |
| Lightning | Conflict | There are conflicting contributions to this softgoal. Some supporting, some against. |

The bilateral interdependencies are attached with information about the contribution of the subgoal to the parent goal. They are annotated next to the contribution arrow symbol. The contributions are explained in Table 2. When evaluating the SIG, the contributions together with the labels of the child softgoals determine the parent softgoals' label. The contribution acts as a multiplier where a negative sign reverses the multiplied value of a label.

**Table 2. SIG contributions**

| Symbol | Contribution | Explanation |
|---|---|---|
| ++ | MAKE | Child label is strongly propagated to parent. |
| + | HELP | Child label is somewhat propagating to parent. |
| = | EQUAL | The two softgoals share the same label. |
| - | HURT | Negated child label somewhat propagating to parent. |
| -- | BREAK | Child label strongly is negated and propagated to parent. |
| ? | UNKNOWN | Interdependency unknown, child does not affect to paren |

### 2.2 Tools for the NFR Framework

There is very little tool support for the NFR Framework in RE. Utilizing the NFR Framework has mostly been reliant upon drawing SIGs manually and further utilizing the resulting NFRs and operationalization decisions separately. However, there is some tool support. The NFR-Assistant [9] is a stand-alone tool that can be utilized to draw SIGs and to derive softgoal labels based on the interdependencies, contributions and operationalization labels. The Softgoal Extension for StarUML [10] provides drawing and evaluating capabilities for SIGs within the open source UML/MDA Platform StarUML [11]. The StarUML extension includes the regular SIG graph features of NFR Framework. Due to its integration to an UML/MDA tool, the NFR outcome can be utilized in software modelling processes.

## 3 NFR+ Framework

We propose extensions to the NFR Framework in order to improve requirements traceability, to encourage the setting of more accurate and beneficial requirements and to enable early and up-to-date verification of requirements. We address specifically

MDD by connecting requirements modelling to software modelling. We call this extended NFR Framework, NFR+ Framework.

## 3.1 Measurable Non-Functional Requirements

To enable verification of NFRs, specific pass/failure criteria in terms of numeric values, metrics and measurement arrangements must be included. We call these measurable NFRs. We utilize a template adapted from [12] to describe measurable NFRs. In [13] it is shown how measurable NFRs can be added to design models in order to define the part of a design model which is responsible of fulfilling a certain individual NFR. In [14], there is a run-time measurement technique described for collecting data and evaluating the performance characteristics of such partial design model. The mechanism utilizes code generation to inject probes into the source code to be tested, which also reports its observations back to the design tool. The run-time measurements can thus notify whether an individual NFR has been fulfilled or not. Such mechanism can be utilized to verify run-time NFRs such as those related to performance issues  For evolution-time non-functional features, e.g., extensibility, methods such as ATAM [15] can be utilized for evaluation. In the case of impartial, unrunnable code or the need for external off-line testing, the measured values can be manually inputted to measurable NFRs after tests have been conducted or when estimates have otherwise been formed. Regardless of the source of information the NFR is evaluated against, it is important to be able to connect this information to the rest of the models in order to be able to trace the impact to the whole system. This is done through meterization which extends the existing NFR framework to take into account the collected empirical verification data.

## 3.2 Meterization

Meterization is a special relationship that connects measurable NFRs to SIG softgoals. Its symbol resembles a gauge that displays the current status of the connected measurable NFR in graphical terms. Figure 2 shows the three different meterization symbols which are fail (left), pass (right) and undefined (centre). Corresponding colour codes are red for fail, green for pass and yellow for undefined. The undefined state exists if the connected measurable NFR has not yet been evaluated within any of the design models.

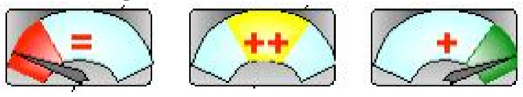


**Figure 2. Meterization relationship symbols.**

Contribution of the meterization symbol to the SIG softgoals is similar to the contributions of softgoals amongst each others with the exception that the EQUAL (=) contribution always overrides any other SIG contributions affecting any other connected softgoal. This emphasizes the empirical verification of defined measurable NFRs so that new tests can automatically alter the SIG state to correctly reflect the observed status of the implementation.

### 3.3 Implementation of NFR+ Framework

The NFR+ Framework was implemented using MetaEdit+ (ME+), which is a commercial DSM tool created by MetaCase. ME+ includes tools to define Domain-Specific Modelling Languages (DSMLs) with GOPPRR (Graphs, Objects, Ports, Roles, Relationships) a metamodeling language and generators with MERL scripting language in addition to providing basic modelling facilities. ME+ also provides an Application Programming Interface (API) which is accessible by all SOAP-enabled (Simple Object Access Protocol) programming languages.

The measurable NFR model entity is defined in the NFR+ metamodel structurally in a requirements model entity which can be connected to SIG graphs with a meterization relationship that also describes the status of the measurable NFR with the meterization symbol. The requirements model entity itself does not contain any measured or otherwise evaluated values because the evaluation and measurement of each requirement type is dependent on the requirement type and application domain. Therefore, the requirement model entity finds special model entities called monitoring mechanisms that should be attached to the application models and finds the test and evaluation results from there. If no monitoring mechanisms are found, the meterization symbol is automatically set at “undecided”. Otherwise, the meterization symbol automatically indicates pass or failure depending on the test or evaluation result. Describing the other SIG concepts of the NFR Framework with ME+ metamodel is straightforward and thus deserves no detailed analysis here.

The SIG evaluation generator was implemented using MERL scripting language. The evaluation generator when initiated traverses the SIG graphs starting from the topmost softgoals. The evaluation is conducted in a recursive manner evaluating the subordinates of each softgoal first and propagating their labels through the interdependency contributions to the parent to eventually determine its new label. The evaluation is based on a predefined, but modifiable, contribution catalogue.

The contribution catalogue is a special SIG graph that defines the outcome of pairs of child labels and contributions. The SIG evaluation generator tries to find this unique graph from the currently active project in the ME+ tooling environment and checks the resulting parent label from the diagram if such is defined. Otherwise, defaults are used according to the original NFR Framework. The contribution catalogue can be modified to influence the evaluation. For example, normally a signal of an Accepted label propagated through HELP contribution results an Accepted label within the parent node. An alternative logic might be that such combination results only a Weak positive, instead of Accepted. If such change is needed, this could be achieved simply by changing one label within the contribution catalogue, the one describing this combination’s result

The SIG graph is updated after the model evaluation with an external application which is generated by the evaluation generator. An external application must be generated since ME+ currently does not allow the alteration of properties of objects within the graphs directly from the MERL scripts. The generated external application consists of Python script that manipulates the SIG graph via the ME+ API. In the end, the script is executed via external command to the Python interpreter and the model is accordingly updated.

## 4 Demonstration of NFR+ Framework in a Laboratory Case

In this section, the usage of NFR+ Framework is demonstrated in a laboratory case of image processing applications. The demonstration illustrates some of the possibilities the NFR+ Framework provides in order to promote requirements traceability.

### 4.1 Functional and Non-Functional Requirements for an Example Image Processing Application

Stream-oriented computing systems are characterized by parallel computing components that process potentially infinite sequences of data [16]. Such systems are common e.g., in embedded systems, digital signal processing, image and video processing and cellular base stations. The primary purpose of such systems is to read data from an input stream, manipulate the data with a filter chain, and forward the manipulated data to a sink. Briefly, the system can be considered to be based on the Pipes- and Filters architecture pattern [17] which consists of a set of data manipulation filters which are connected together with pipes. The filters do not share the state thus the filters can be connected together in arbitrary order as long as the semantics are correct.

As a laboratorial case of stream-oriented computing systems, a subsystem of an image processing application, i.e. video camera, is utilized. The main responsibility of the example subsystem is to flip incoming 1Mpixel images 90 degrees to the right to match the tilt of a display which is utilized for displaying input images. The input images are also required to have sepia toning. The sepia-converted images must also be compressed to JPG format and saved to temporal data storage from where the images are forwarded to the next subsystem. NFRs for the subsystem are as follows:

- Average frame rate of the display should be 25fps with good video quality.
- Average throughput of the subsystem should be more than 10fps in average.
- Images produced by the subsystem should be of adequate quality.

### 4.2 NFR+ Softgoal Interdependency Graphs

The functionalities can be divided into four task specific functional blocks, i.e. *RotationFilter*, *SepiaConverter*, *Display* and *TemporalStorage* which is responsible for JPG conversion and saving the manipulated images to data storage. There are also *Resize* filters available for altering the resolution. The functional blocks form topics for the SIG.

According to the preceding functional requirements and NFRs, the SIG presented in Figure 3 can be drawn. Overall subsystem satisfaction depends on the video quality the *Display* should provide, subsystem throughput and the image quality the subsystem forwards. The *Display* image quality depends on throughput of *SepiaConversion*, *Rotation*, and resolution of the viewed images where the priority is on frame rate. As presented, throughput of the filters, image compression and saving of the images into *TemporalStorage* are heavily affected by the image resolution. However, the image resolution affects the overall system satisfaction positively in addition to *Display* quality, therefore there is a distinct trade-off between throughput and image quality.

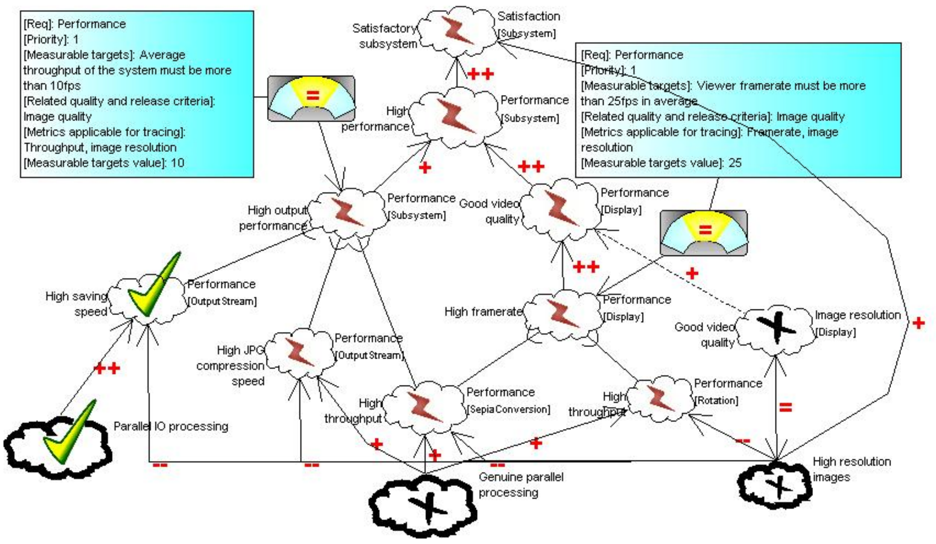

[Req]: Performance
[Priority]: 1
[Measurable targets]: Average throughput of the system must be more than 10fps
[Related quality and release criteria]: Image quality
[Metrics applicable for tracing]: Throughput, image resolution
[Measurable targets value]: 10
=
Satisfactory subsystem
Satisfaction [Subsystem]
++
High performance
Performance [Subsystem]
[Req]: Performance
[Priority]: 1
[Measurable targets]: Viewer framerate must be more than 25fps in average
[Related quality and release criteria]: Image quality
[Metrics applicable for tracing]: Framerate, image resolution
[Measurable targets value]: 25
+
++
High output performance
Performance [Subsystem]
Good video quality
Performance [Display]
=
High saving speed
Performance [OutputStream]
++
+
High framerate
Performance [Display]
Good video quality
Image resolution [Display]
+
High JPG compression speed
Performance [OutputStream]
++
High throughput
Performance [SepiaConversion]
High throughput
Performance [Rotation]
=
Parallel IO processing
+
+
--
+
--
--
--
Genuine parallel processing
High resolution images

**Figure 3. SIG for the example application.**

As presented in the SIG, throughput of the overall subsystem can be improved by enabling a parallel image saving process as well as by enabling genuine parallel processing of the filters and compression. Since Python 2.5 does not currently support genuine parallel processing with standard threads, parallel processing of multiple images cannot be applied. However, threads can be applied to enhance IO processing.

The SIG is also accompanied by specific measurable NFRs as depicted in Figure 3. The subsystem output performance subgoal is accompanied by an NFR that explicitly states that the throughput should be more than 10fps on average. Similarly, the frame rate of the *Display* is explicitly defined with the measurable NFR model entity. The connecting meterization symbols indicate the current status of measurable NFRs. With the SIG being constructed, it can be evaluated to specify labels of the softgoals. The evaluated SIG in figure 3 reveals that the subsystem does not satisfy its NFRs. The major problem with throughput of the filters seems to be caused by the lack of possibility for genuine parallel processing. However, it may be possible that the filters in serial processing mode might satisfy the requirements, thus the subsystem needs to be tested.

## 4.3 Application Modelled with Extended M-Net

### 4.3.1 The Modelling Facilities

In this paper, we use a refined version of the earlier developed M-Net modelling language [13][14] which enables modelling of simulations of parallel computing filter chains in the domain of stream-oriented computing systems. We have extended the M-Net with the capability for modelling image processing systems. The extended M-Net uses real image processing filters instead of simulations used in previous version. Otherwise the languages are the same. For M-Net, complete Python source code generation from models was developed including support for image processing. As a graphic library, the Python Imaging Library [18] was used.

Considering the testing of NFRs, the application models can be included with NFRs and monitoring mechanisms [13]. The monitoring mechanisms can be connected to those parts of interest in the application models to enable the monitoring of throughput of images i.e. the amount of images handled per second. The generated code is embedded with monitoring mechanisms that enable run-time reporting back to the application model via ME+ API.

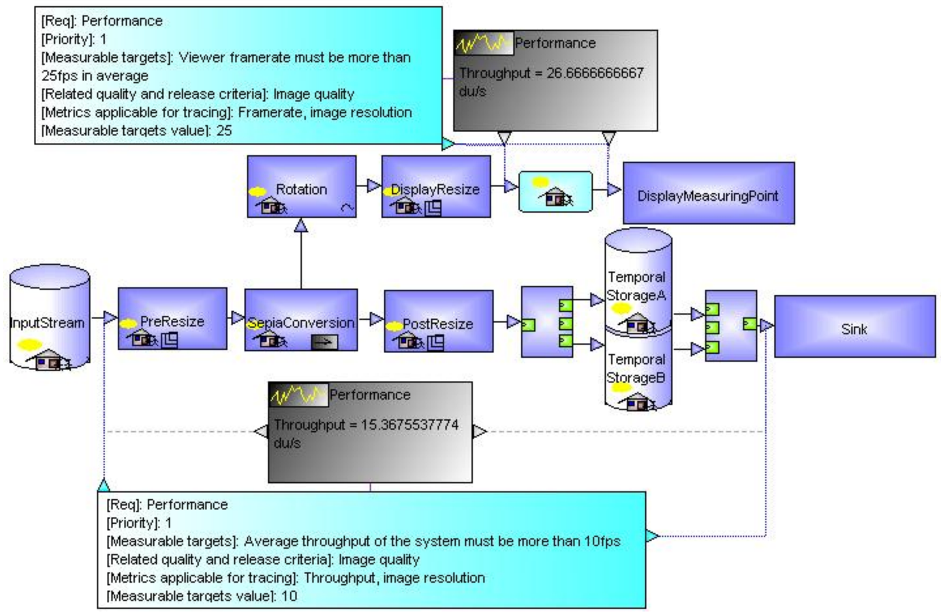
### 4.3.2 The Application Model

An application model that satisfies the functional requirements is presented in Figure 4. *InputStream* functions as a data source of bitmap images which are forwarded to *PreResize* filter. *PreResize* drops the image resolution by half thus causing a significant impact on image and video quality. This implements not using high resolution operationalization.

Resized images are forwarded from *PreResize* to *SepiaConversion*. Then the stream divides into two paths. The path to *Display* (the topmost path) leads to *Rotation* which rotates the images 90 degrees to the right and forwards the image to *DisplayResize* to decrease the image resolution even more to maximize the frame rate.

After sepia conversion in the lower part of the subsystem, the images are resized back to the required 1Mpixel by *PostResize*. The images are further forwarded to a *switch* which forwards the first input image to the first *TemporalStorageA* which immediately starts processing the data. The second image is forwarded to the *TemporalStorageB* which starts computing the image parallel to the previous data storage. This way the data storages receive every other image and therefore halve the amount of images to be handled per data storage. This solution is an implementation of parallel IO processing operationalization defined in the SIG. After image compression and saving, *TemporalStorages* forward the data to the *Sink*.

The measurable NFRs are presented also in the finalized application model (Figure 4) and connected to the corresponding parts in the model to maintain a trace link between the NFRs presented in Section 4.1 and the application model. The application model is also accompanied by appropriate monitoring mechanisms connected to the measurable NFRs. After generating an executable and executing it, the application reports the measured values back to the application model. As shown, the frame rate of the *Display* is 26.7fps (see the dark rectangle on the top) and the throughput of the overall system is 15.4fps. It seems that the application now satisfies these requirements.


[Req]: Performance
[Priority]: 1
[Measurable targets]: Viewer framerate must be more than 25fps in average
[Related quality and release criteria]: Image quality
[Metrics applicable for tracing]: Framerate, image resolution
[Measurable targets value]: 25

Performance
Throughput = 26.6666666667 du/s

Performance
Throughput = 15.3675537774 du/s

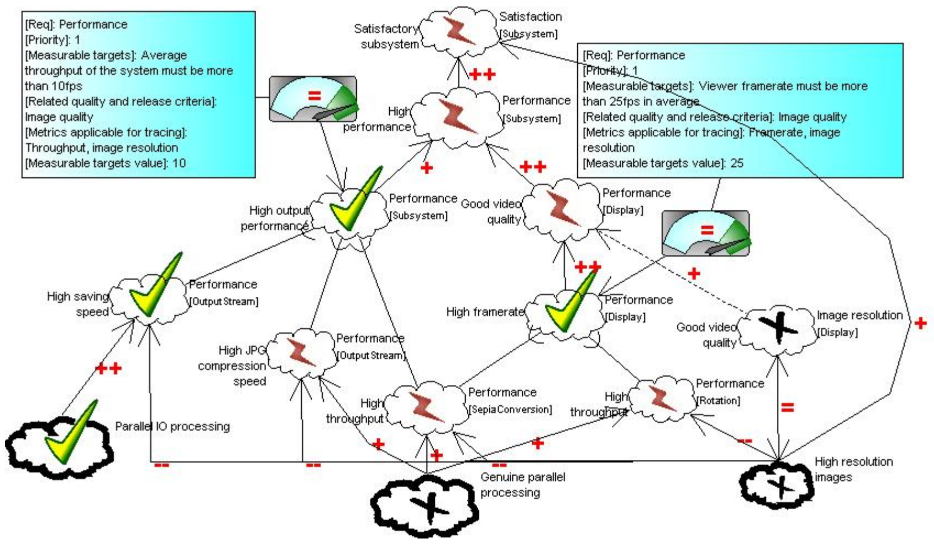
[Req]: Performance
[Priority]: 1
[Measurable targets]: Average throughput of the system must be more than 10fps
[Related quality and release criteria]: Image quality
[Metrics applicable for tracing]: Throughput, image resolution
[Measurable targets value]: 10

**Figure 4. Image processing application modelled with M-Net.**

## 4.4 The Analysis of NFR+ Softgoal Interdependency Graphs

After the application has been modelled, generated and executed, analysis of the test results to the overall subsystem can be performed by scrutinizing the automatically updated SIG (see Figure 5). As the overall subsystem throughput and *Display* frame rate are satisfactory, these are represented as “pass” meterization symbols. The impact of the measured performance values to the overall subsystem can be discovered by evaluating the SIG.

The measured values override previous conflicts but still there is a conflict in the “Good video quality” softgoal whose label is also propagated to overall subsystem satisfaction. This conflict is clearly caused by using low-resolution images as input to the *Display*. At this stage the requirements engineer and the application modeller should try to find a solution for the problem. The SIG clearly reveals that the only way to satisfy the overall subsystem satisfaction requirements is using higher resolution images. As discussed above, this on the other hand has a drastic impact on overall throughput. Thus, performance of the filters should be increased when using high-resolution images. If no solution can be found, the impact of the failure should now be traced to the other SIGs and the overall impact at the system level should be evaluated.


[Req]: Performance
[Priority]: 1
[Measurable targets]: Average throughput of the system must be more than 10fps
[Related quality and release criteria]: Image quality
[Metrics applicable for tracing]: Throughput, image resolution
[Measurable targets value]: 10

[Req]: Performance
[Priority]: 1
[Measurable targets]: Viewer framerate must be more than 25fps in average
[Related quality and release criteria]: Image quality
[Metrics applicable for tracing]: Framerate, image resolution
[Measurable targets value]: 25

**Figure 5. SIG with test results.**

## 5 Discussion

The presented NFR+ Framework addresses the challenges of bridging the gap between RE and verification in a transparent fashion observable also through the SE phases. The original NFR Framework provides a solid and expressive framework for eliciting NFRs and binding them to functional requirements. However, the NFR Framework is mostly utilized only the elicitation process of the NFRs. To bridge the gap between requirements engineering and software engineering it is required that the SIGs in the NFR Framework are continuously taken into account in SE but that also the outcomes of SE should be utilized in RE. Our extension to the NFR Framework strives for just that.

As presented, during SE, the elicited NFRs and operationalizations offer information about the design rationale. The measurable NFRs define a concrete instantiation for an abstract softgoal definition that cannot be, as such, usable for design input nor for verification purposes. The measurable NFRs represent additional and optional concretization in parallel with operationalizations. However, the direction of information is two-way. Whereas operationalizations are design specifications, measurable NFRs relay information to both directions to and from RE workspace. Measurable NFRs include concrete verification criteria, but also reflect the status of their realization in the SIGs representing all of the NFRs. As the dimension of adding concretization and information is orthogonal with operationalizations and functional requirements, we believe that this is an important step forward to closing the gap between RE and SE.

The next step towards completely unified MDD-environment should include importing existing requirements from other tools into the SIGs and SW modelling environments automatically in order to avoid redundant work and management overheads. While the NFR+ Framework offers full tooling for starting a new project from scratch, there might be a vast requirements database in many realistic cases that are inherited from earlier products or versions. For practical reasons synchronizing with them would provide a great benefit. We plan to study this in the future.

Further studies are yet needed to find out the applicability of our approach in large-scale real cases. For example managing the complexity of multiple, large design models and vast SIGs should be practically evaluated.

## 6 Conclusion

In this paper an extended version of the NFR Framework, which we call the NFR+ Framework, was presented. In addition, tooling with MetaEdit+ language workbench for the framework was also presented. We have extended the NFR Framework by adding a concept of measurable NFR. Measurable NFRs can provide evidence-based information about achieving the preset or currently chosen NFR goals. They also serve as a connecting point and guide to software designers by stating the desired outcome.

This linkage between requirements and design provides an approach for joining the separate areas of requirements engineering and software engineering. The solution serves as a communication tool between stakeholders by offering a formal common language and documentation. The NFR Framework also per se promotes the refinement of the requirements needed during elicitation into such a form that the design can be directly justified by them. In addition, the measurable NFRs guide the requirements engineers in their aim to create measurable requirements which will also serve as verification and thus provide overall quality monitoring view to the system under development. The traceability of requirements becomes transparently visible all the way from code-implementing design models to the highest level NFRs and vice versa.

We demonstrated our approach with a laboratory case and illustrated how the NFR+ can help in detecting problematic designs. We also showed how the interlinkage helps in verification of the NFRs during the whole time span of software development; from forming first SIGs to finally testing related implementations. This verification is based on connecting measurable NFRs to SIG through the meterizations.

## ACKNOWLEDGMENTS

The work presented here has been developed within the MoSiS project ITEA 2 – ip06035. MoSiS is a project within the ITEA 2 –Eureka framework.

## References

[1] Cheng, B. and Atlee, J.: Research directions in requirements engineering. In FOSE ’07: 2007 Future of Software Engineering, Washington, DC, USA, 2007. IEEE Computer Society, pp. 285–303

[2] Baudry, C., Nebut, C. and Traon, Y. L.: Model-Driven Engineering for Requirements Analysis. In Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC), 2007, pp 459-459.

[3] Matinlassi, M. and Niemelä, E.: The Impact of Maintainability on Component-based Software Systems. In 29th Euromicro Conference (EUROMICRO’03), Turkey, 2003, pp. 25-32.

[4] En-Nouaary, A., Khendek, F. and Dssouli, R.: Fault Coverage in Testing Real-Time Systems. In Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA ’99), Hong Kong, 1999, pp. 150-157.

[5] Graham, D.: Requirements and testing: seven missing-link myths. In IEEE Software, 19(5), 2002, pp. 15-17.

[6] Chung L., Nixon, J. M. B. and Yu, A.: Non-functional Requirements in Software Engineering. Springer, Reading, Massachusetts, 2000.

[7] Robertson, S.: An Early Start to Testing: How to Test Requirements. In EuroSTAR ’96, Amsterdam, December 2-6, 1996.

[8] Kelly, S. and Tolvanen, J-P.: Domain-Specific Modeling – Enabling full code generation. John Wiley & Sons, New Jersey, 2008, 427p., ISBN: 978-0-470-03666-2.

[9] Tran, Q. and Chung, L.: NFR Assistant: Tool Support for Achieving Quality. In ASSET ’99, 1999, pp. 284–289.

[10] Supakkul, S.: The Softgoal UML Profile: An NFRs Modeling Tool. URL: http://www.utdallas.edu/~supakkul/tools/softgoal-profile/softgoal-profile.html [Visited July 2009].

[11] StarUML, StarUML - The Open Source UML/MDA Platform. URL: http://staruml.sourceforge.net/en/ [Visited June 2009].

[12] Ebert, C.: Putting requirement management into praxis: dealing with nonfunctional requirements. Information & Software Technology 40(3): 175-185, 1998.

[13] Merilinna, J. and Räty, T.: A Tooling Environment for Quality-Driven Model-Based Software Development. In 9th OOPSLA Workshop on Domain-Specific Modeling, Orlando, Florida, USA, 2009.

[14] Merilinna, J. and Räty, T.: Bridging the Gap between Quality Requirements and Implementation. In The Fourth International Conference on Software Engineering Advances (ICSEA 2009), September 20-25, 2009 - Porto, Portugal.

[15] Kazman, R., Klein, M. and Clements, P.: ATAM: Method for architecture evaluation. Carnegie Mellon University, Software Engineering Institute, Tech. Rep. CMU/SEI-2000-TR-004 ESC-TR-2000-004, 2000, 83 p.

[16] StreamIt, Research overview page, URL: http://www.cag.lcs.mit.edu/streamit/shtml/research.shtml [Visited July 2009]

[17] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M.: Pattern-oriented software architecture – a system of patterns. Chichester, New York: Wiley, 1996, 457 p.

[18] PythonWare, Python Imaging Library: URL: http://www.pythonware.com/products/pil/ [Visited July 2009]