Supporting Extra-Functional Properties Preservation in Model-Driven Engineering of Embedded Systems

Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin

School of Innovation, Design and Engineering
Mälardalen Real-Time Research Centre
Mälardalen University, Västerås, Sweden
firstname.lastname@mdh.se

MRTC report ISSN 1404-3041 ISRN MDH-MRTC-257/2011-1-SE

Abstract. In order for model-driven engineering to succeed, automated code generation from models through model transformations has to guarantee that extra-functional properties modelled at design level are preserved at code level. A full round-trip engineering approach could be needed in order to evaluate quality attributes of the embedded system by code execution monitoring/analysis tools and then provide back-propagation of the target code analysis results to modelling level. In this way, properties that can only be estimated statically are evaluated against runtime values and this consequently allows to optimize the design models for ensuring preservation of analysed extra-functional aspects. This paper presents an approach to support the whole round-trip process starting from the generation of source code for a target platform, passing through the monitoring of selected system quality attributes at code level, and finishing with the back-propagation of measured values to modelling level. The technique is validated against an industrial case-study in the telecommunications applicative domain.

1 Introduction

The increasing complexity of modern software systems demands adequate development techniques able to reduce the complexity of the problem, allow to focus on the aspects that matter in the design of the application, and permit to reason about the scenario in terms of domain-specific concepts. In this respect, Model-Driven Engineering (MDE) vision relies on facilitating the system development by creating, maintaining and manipulating models that provide abstractions of a real phenomena with an intended purpose [1]. Rules and constraints for building a model have to be properly stated through a corresponding language definition. In this respect, a meta-model describes the set of available concepts and well-formedness rules a correct model must conform to [2]. A system is developed by refining models through model transformations starting from higher and moving to lower levels of abstraction until code is generated. A model transformation

converts a source model to a target model preserving their conformance to the respective meta-models [3].

One of the major ambitions of MDE is to provide automated code generation to be executed on specific target platforms; however, such a goal is too often seen as the final and non-coming back step of an MDE approach [4]. Therefore, quality in terms of extra-functional properties of the system modelled at abstract levels may not be preserved at code level since many of such aspects can not be predicted without code execution [5]; that is the reason for which such properties need to be computed at code level through monitoring activities [6]. Then, in order to be able to perform an evaluation of expected against computed extra-functional properties, the model-code abstraction gap must be settled; this can be performed by propagating the extra-functional computed values back to the source models. Hence, we hereby claim that the generation of code and its execution on target platforms should be rather seen as a transitional step in the development; the results coming from the execution would be utilized as an enrichment of the design models for further extra-functional evaluation. Eventual optimization of such models can then be performed in order to generate code with preservation of the desired extra-functional properties.

This article discusses motivations and challenges in providing an automated round-trip engineering support for MDE of embedded systems with focus on ensuring extra-functional properties preservation throughout the entire process, that is to say from modelling to code level. In fact, embedded systems' resources limitation stresses the criticality of extra-functional properties measurement at code level. Moreover, this work proposes a solution to the implementation of such automated round-trip engineering mechanism: generation of source code entails the creation of trace links between models and code, code execution is monitored, and detected values are appropriately back-propagated to modelling level. In this way, a complete extra-functional evaluation of the source model can be enabled, and the automation in the preservation process relieves the developers' effort in shouldering the heavy burden (e.g. additional testing activities on models, involvement of domain experts) of manual activities in that direction.

Despite other solutions for the generation of *full*[1] implementation code exist, its combination together with the provided cross-cutting back-propagation capabilities make the contribution of this paper unique. In fact, a process combining the features provided by the solution we propose has not been fully exploited yet in the current state-of-the-art.

In order to inspect significant characteristics of the approach, such as scalability and reusability, the proposal has been validated against a running case-study on top of the CHESS framework [7]; the target system is a partial version of the Asynchronous Transfer Mode (ATM) [8] Adaption Layer 2 (AAL2) industrial subsystem which was originally intended to adapt voice for transmission

[1] In our proposal and in the remainder of the paper, code that can be directly executable right after its automatic generation without any further modification is addressed as *full* or *100%*

over ATM and is currently used in telecommunications as part of connectivity platform systems.

The rest of the paper is structured as follows. Section 2 identifies the motivation that led us to the definition of the proposed approach and which aspects have already been partially explored in the current state-of-the-art. In section 3 the proposed approach is described and fully unwound in its details together with challenges and solutions for each step of the process. The running case-study, introduced in section 4, is used to validate the approach in sections 5 and 6 where the actual implementation of the proposed solution is reported too. In section 7 a final discussion is presented while conclusions and planned future enhancements are proposed in section 8 and 9.

2 Background

The integration of model-driven and component-based (MDCB) processes is meant to help in handling the ever-increasing complexity of embedded software systems design. In particular, such integration discloses opportunities to reduce costs and risks by: (i) enabling effective modelling of extra-functional properties such as safety, reliability, availability and dependability, to mention a few, and (ii) providing automation where applicable in the development process [9]. In the last years, MDCB has been recognized as extremely promising and the large number of works recently published on this subject gives proof of this research trend [10]; tools and frameworks have been developed for supporting such development process [11–13].

Our approach is placed within a MDCB process and focuses on providing round-trip support for aiding in ensuring a required level of quality preservation in terms of extra-functional properties from the modelling artefacts to the generated code. Preservation of those properties throughout the development process by means of appropriate description and verification is of paramount importance since it allows to reduce final product verification and validation effort and costs by providing correctness-by-construction. In fact, managing extra-functional properties by means of design and evaluation at early development phases allows the developing team to have control over the preservation of such properties at product level [14, 15].

In the embedded domain resources limitation sharpens the criticality of having extra-functional properties evaluation at code level: for instance, when dealing with resources consumption or execution time, it is generally only possible to provide estimates of values boundaries and derive statistical behaviours of their variations. As a consequence, it is not possible to have precise values until the design is run in terms of code on a specific platform thereby some of the information and values used for validating the design can only be gathered at code level. Despite early simulation activities can help in evaluating a certain set of properties [16] as well as solving specific issues such as bottlenecks and deadlocks, there still exist extra-functional properties that can not be precisely evaluated at modelling time. Therefore, this work aims at providing support for such properties and leaves apart simulation and analysis activities performable

at modelling level. Precise values are obtained only when the design is run in terms of implementation code on a specific platform. Then, a proper round-trip support for enabling back-propagation of such values to the source model is of critical relevance in order to aid the MDBC process in achieving automated generation of code with preservation of extra-functional properties across all the abstraction layers.

Generally, management of extra-functional properties is considered a core development task; in the embedded domain such activity has still to be improved [17]. Efforts in dealing with extra-functional properties of composition can be found in the Web Service domain [18, 19]; domain-specific languages and UML profiles have been defined to model extra-functional properties [20, 21]. Results in the direction of ensuring real-time properties by the definition of a MDCB approach can be found in the results of the ASSERT project [22]; nevertheless, effective solutions for composability with guarantees in embedded real-time domain are still missing. Other attempts propose to solve the problem by back-propagation, i.e. by reporting the measured values back to the models in order to possibly fix and/or refine estimated numbers. Navabi et al. in [23] in the early 90's, and some years later Mahadevan and Armstrong in [24], came up with different approaches for back-annotating behavioural descriptions with timing information; however, both operate horizontally[2] in terms of abstraction levels and no automation is provided. It is worth noting that being able to automatically annotate the source model with monitored values is of critical importance in order to avoid the developer to inspect the generated code for understanding the relationships between measures and model entities at design level.

In the literature, Varró et al. propose in [25] back-propagation for enabling execution traces retrieved by model checkers or simulation tools to be integrated and replayed in modelling frameworks; even though some similarities to our approach might be found when dealing with traceability issues, the two approaches aim at solving two different problems. The most similar approach to ours is described by Guerra et al. in [26] where back-propagation of analysis results to the original model by means of triple graphical patterns is described. Nevertheless, the approach is meant to horizontally operate at modelling level with propagation of data among models. While, dealing with embedded real-time systems, our approach focuses on vertically propagating analysis results computed at code level back to design models for better understanding of those extra-functional properties that can not be directly measured at higher levels of abstraction.

The concept of traceability in software development is generally referred to as a mean to store relationships between interconnected artefacts [27]; with respect to MDE, since model transformations are the mean to operate on models, trace links are often used to keep track of model transformation results, that is to relate source elements with corresponding target entities together with

[2] In the paper, horizontal and vertical are used for specifying the direction of data transitions among artefacts either at the same (e.g. horizontal) or at different (e.g. vertical) level of abstraction

the rules involved in their creation. Typically, trace links are exploited for synchronization purposes or to evaluate the impact of source modifications to the existing generated targets [28]. This paper extends such concept to keep track of implementation code portions corresponding to design model elements in order be able to update their quality attribute values coming from system execution monitoring.

System monitoring is a widespread activity to survey applications' behaviour, especially in those cases where systems' failures can have serious impacts or even catastrophic consequences [6]. In general monitoring tasks are devoted to check systems attributes to understand their quality [29], or to detect behavioural patterns requiring runtime adaptations [30]. In this respect, this work exploits monitoring results to refine design models with sharpen quality attributes values detected during the execution of the system. Such precision is required for embedded systems, since resources utilization is of critical importance for a successful design of the application.

3 A Generic Round-trip Support for Preservation of Extra-Functional Properties

This work illustrates a general technique whose goal is to provide support in terms of traceability and back-propagation features for MDCB processes to achieve preservation of those extra-functional properties which can not be accurately predicted statically at modelling level, but rather measured at code level. By adhering to the MDE vision, implementation code for the platform taken into account is generated from the source models; then, by monitoring the execution on the target platform, computed values are automatically propagated back to the modelling level. The aims are multiple: at an initial development stage, estimated values can be iteratively refined through executions of the generated system and corresponding monitoring/back-propagation activities; in turn, design models can be evaluated and possibly refined to achieve extra-functional properties preservation and hence correctness-by-construction [5] for the generated implementation; in the same way, resource optimization can be enabled through a better resource utilization based on actual values.

In order to achieve extra-functional properties preservation and hence correctness-by-construction, the generated code is not meant to be edited by hand. Possible optimizations are indeed not performed directly through code editing, but rather by re-iterating the code generation process once the design models have been refined according to the evaluation of their extra-functional properties made possible by the back-propagation.

In the following, the proposed approach is decomposed in three fundamental issues, namely how to generate and store trace links between source models and generated code, how to retrieve useful information from monitored code execution results, and how to propagate the computed values back to the source models; for each of them, challenges and proposed solutions, together with their application to a running case-study, are discussed.

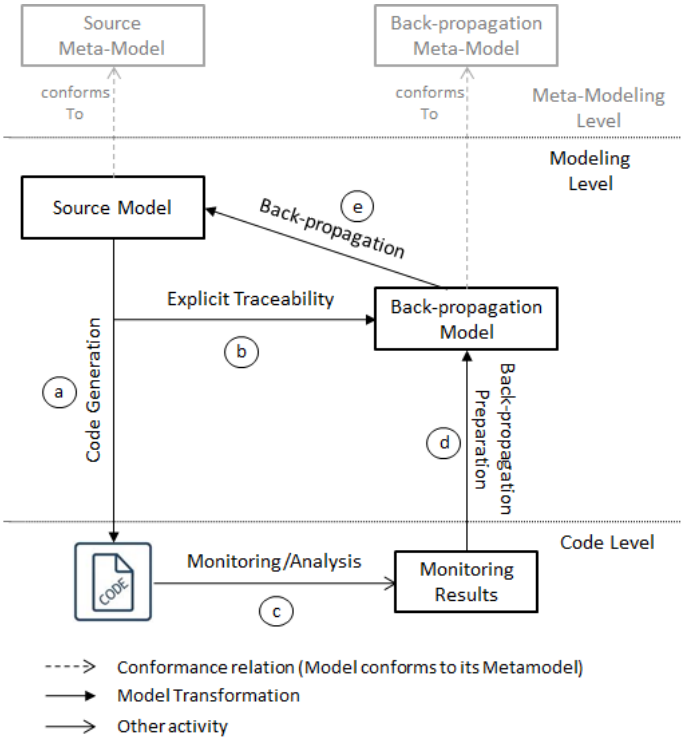

Fig. 1. Round-trip Support

The general goal of this work is to propose a round-trip support which aims at solving the challenge of generating implementation code ensuring that extra-functional concerns modelled at design level are preserved at code level (Fig. 1). Once design modelling tasks have been successfully completed, the objective is to enable automatic generation of implementation code from source models. Taking design models as source artefacts[3], we generate target code through appropriate model transformations (Fig. 1a). Information regarding tracing of source (e.g. model elements) and target (e.g. code segment(s)) artefacts has to be defined and maintained for further back-propagation activities. Therefore, code generating transformations have to be properly defined by encoding apposite rules for the generation of traceability links (explicit traceability [31]) between models and code (Fig. 1b); such rules populate the back-propagation model with traceability information according to the meta-model depicted in Fig. 2. Once the code has been generated as well as the traceability links, quality attributes of the system can be evaluated by selected code execution monitoring/analysis tools (Fig. 1c). Depending on the capabilities of such tools and their output format,

[3] In the remainder of the paper source and design models refer to the artefacts at the highest level of abstraction as shown in Fig. 1

different actions, varying from text-to-model to model-to-model transformations (Fig. 1d), are required to extract and formalize execution results in order to have a completely traced information chain from models to monitoring results. The last step of the round-trip approach aims at finally annotating the source models with the code execution results (Fig. 1e) through dedicated model-to-model transformations.

As depicted in Fig. 1, by containing information concerning both traceability and monitoring results, the back-propagation model can be considered the core artefact in the process. That is to say that, as long as the modelling language does not change, the transformations performing the back-propagation will not need to be modified, even if different tools are used for monitoring activities. An alternative solution could indeed be to back-propagate the monitoring results directly to the model with the support of a simpler traceability model; in this case for each monitoring tool a different back-propagating transformation should be provided. Using an intermediate back-propagation model reduces the approach adaptation overhead caused by the adoption of different monitoring tools, since only ad-hoc transformations from monitoring results to the back-propagation model, which are generally much less intricate than transformations from monitoring results to source models due to modelling languages' complexity, have to provided.

3.1 Code Generation and Traceability

In the proposed approach, the task of automating the generation of implementation code does not only concern the actual transformation from design models to code since tracing information between model elements and generated code segments has also to be defined for enabling back-propagation activities. Traceability can be any relationship existing between artefacts within a software engineering life cycle. These relationships include: (i) explicit links derived from for/back-ward transformations, (ii) links derived from code analysis, (iii) inferred links computed on the basis of change management of system's items [27].

Our approach relies on models and transformations as main artefacts; models represent the system at different levels of abstraction and transits back and forth among these levels are usually achieved through transformation of such models. Therefore definition and maintenance of traceability links to cope with consistency among models, code and transformations are crucial. That is the reason for which model transformations in charge of code generation must be properly defined by encoding apposite rules for the generation of explicit traceability links between source and target. In this way, information exchanged among models through transformations are formally stored and maintained in structures that are easily and univocally navigable and information pieces reachable following precise patterns. The most natural structure for such purpose in our case is a model, here called *back-propagation model*, which conforms to the *back-propagation meta-model* depicted in Fig. 2; a slightly similar structure for storing tracing information can be found in [32]. The back-propagation meta-model has been defined for enabling the creation of back-propagation models which are able

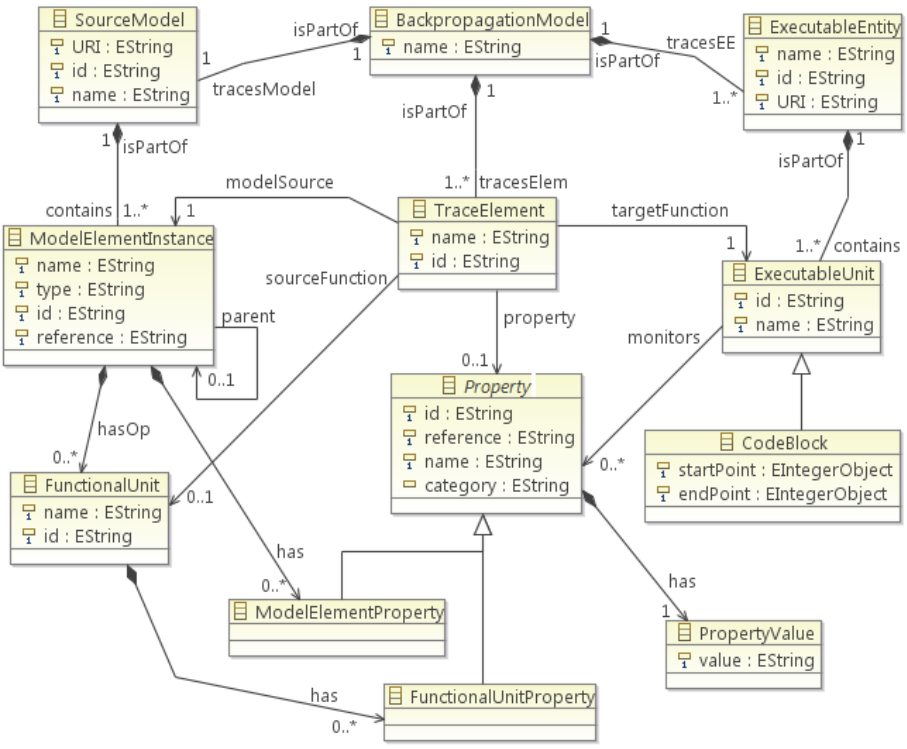

Fig. 2. Back-propagation Meta-Model

to store the information gathered during code generation and monitoring tasks in a structured manner.

Conceptually, two classes of information are stored in the back-propagation model: (i) traceability information and (ii) monitoring results. Traceability information is composed by trace links identified during the code generation task and stored in terms of trace elements between model elements and code. The core concept in the back-propagation meta-model is indeed the trace element, which allows the navigation through all the stored information needed for back-propagating the source model with monitoring results. A trace element *TE* (TraceElement in Fig. 2) can store two different granularities of traceability:

- **Model element level**: in this case it is represented as a triple <*ME, EU, MEP*> where *ME* is a model element (*ModelElement* in Fig. 2) contained in a source model *SM* (*SourceModel* in Fig. 2), *EU* is an executable unit (*ExecutableUnit* in Fig. 2) contained in an executable entity *EE* (*ExecutableEntity* in Fig. 2), and *MEP* (*ModelElementProperty* in Fig. 2) is an extra-functional property defined for *ME* and calculated by monitoring the execution of *EU*. A typical case for such granularity is the component level in a component-based architecture;

– **Model element's functional unit level**: in this case it is represented as a quadruple <*ME, FU, EU, FUP*> where *ME* and *EU* represent the same information as for the model element level. The further level of granularity is maintained by *FU* which represents an operation/method (*FunctionalUnit* in Fig. 2) defined in the model element specification *ME* and *FUP* (*FunctionalUnitProperty* in Fig. 2) which represents an extra-functional property defined for *FU* and meant to be calculated by monitoring the execution of *EU*. A typical case for such granularity is the operations level in the component definition in a component-based architecture.

More generally, the back-propagation model (conforming to the back-propagation meta-model) *BM* (*BackpropagationModel* in Fig. 2) is a triple <*TE**, *SM*, *EE**>, where *TE** is a non-empty set of trace elements, *SM* is a source model (i.e. a composition of model elements *ME* and functional units *FU*), and *EE** is a non-empty set of executable entities which are in turn composed by executable units *EU*. Depending from the code generation and the monitoring activities, an executable unit could even be more specifically defined as a code block with start and end point within the code file (i.e. the executable entity).

Apart from the traceability links, the back-propagation model has to be able to host the information extrapolated from the monitoring activities and that would complete the information chain model-code-results needed for propagating the monitored properties computed values back to the source model. In the back-propagation model, each defined property *P* has one property value *V* (*PropertyValue* in Fig. 2), which is calculated during monitoring activities and represents the value to be propagated back to the related extra-functional annotation's placeholder in the source model.

3.2 Extra-Functional Properties Evaluation

Once the target code has been automatically generated as well as the traceability links, quality attributes of the system can be evaluated by executing the code on a specific platform according to appropriate monitoring routines. At this point, independently from the analysis or monitoring tool/technique used for the measuring activities, the extra-functional properties can be evaluated comparing their expected and measured values; for this reason the latter are to be propagated back to design models. In order to perform such back-propagation we need to be able to navigate through the development artefacts from design models down to monitoring results passing through the code. Therefore defining and maintaining explicit traceability between models and generated code is only the first ring of the traceability chain needed in our round-trip approach. In fact, traceability between code segments and monitoring results has to be defined and maintained too. Therefore, additional actions to manipulate monitoring tools' output results and to properly structure them for easing back-propagation activities are needed.

Back-propagation of monitoring results to modelling level represents the last step of the approach, crucial for evaluating and consequently optimizing the design models for ensuring preservation of analysed extra-functional properties.

For the provision of such capability, the approach fights against well-known reverse engineering challenges in mapping data models derived from data analysis to more abstract conceptual design levels by supporting iteration of the process and bidirectional mapping from models to analysis data models and vice versa [33]. Back-propagation is performed through appropriate model-to-model transformations which enrich the source models with the values of extra-functional attributes gathered at code level by monitoring activities.

This process can be decomposed as follows:

- *Monitoring results representation*: results coming from the monitored execution of the generated code are part of the source artefacts for back-propagation to the design models; the representation format of this information is pivotal. Monitoring results should be maintained into formal structures in order to be fed to the back-propagating transformations. The proposed solution provides storing structures as part of the back-propagation meta-model;
- *Tracing information management*: giving monitoring results as source for the back-propagating transformations is not enough. In fact, the traceability chain defined and maintained along the path from designed models to monitoring results is also part of the source artefacts to be fed to the transformations in order to correctly propagate results back to the corresponding model elements. Depending on the decisions taken when defining traceability, actions to manipulate it and feed it to the transformations will be needed. As well as for the monitoring results, tracing information is maintained using the structures provided as part of the back-propagation meta-model;
- *Annotation of design models*: the very final step of our approach is the actual enrichment of source models with information concerning computation of extra-functional properties derived from code execution monitoring activities. The enrichment should be performed by injecting the computed property values into the related model elements' placeholders at modelling level for completing the design models. The effort to be put in such injection depends on the modelling language's capabilities in modelling extra-functional properties.

Once completed, the back-propagation activity produces an extra-functionally enriched version of the design models. At this point it is possible for the developers to evaluate such enriched models and finally perform further optimization activities on them if needed. The process might require multiple iterations in order to reach the desired quality level, in terms of extra-functional properties, required by the system specification.

4 A Case-study: the AAL2 subsystem

The round-trip support had been preliminarily validated on the Advanced Cruise Control subsystem on top of the PrIDE platform using the ProCom component model [34] considering code generation and traceability links creation as black-box activities. In order to perform a further validation of the approach and

evaluate crucial characteristics such as scalability and reusability, we tested it on a notably more complete case-study on top of a different platform and we defined apposite transformations for generating 100% of the implementation code as well as the traceability information. The target system is a simplified version of the AAL2 subsytem which is well-known in the telecommunication domain as part of connectivity platform systems.

4.1 The CHESS Framework

The cross-domain Composition with Guarantees for High-integrity Embedded Software Components Assembly (CHESS) modelling language and framework [7] are used for the specification of the AAL2 subsystem. CHESS allows the specification of a system together with relevant extra-functional properties such as predictability, dependability and security. Moreover, it supports a development methodology expressly based on separation of concerns; distinct design views address distinct concerns. In addition, CHESS actively supports component-based development. The CHESS component model is conceived in a manner that permits to domain-specific needs to be addressed by adding specialization features to a domain-neutral core. In this manner CHESS intends to support a variety of application domains, the common character of which is to embrace model-driven engineering solutions for the development of dependable and predictable real-time embedded systems. According to the CHESS methodology, functional and extra-functional characteristics of the system are defined in specific separated views as follows:

- Functional: the development style follows component-based design, by which each component is equipped with provided and required interfaces realized via ports and with state-machines and other standard UML diagrams to express functional behaviour. Moreover, the Action Language for Foundational-UML (ALF)[35] is used to enrich the behavioural description; in this way, we reach the necessary expressive power to be able to generate 100% of the implementation code directly from the functional models with no need for manual modification of the code after its generation;
- Extra-functional: in compliance with principle of separation of concerns adopted in CHESS, the functional models are decorated with extra-functional information thereby ensuring that the definition of the functional entities is not altered.

4.2 Modelling the AAL2 Subsystem

The structure of the system (Fig. 3) is composed by three main components: (i) *NCC*, (ii) *AAL2RIClient*, (iii) *NCIClient*. Each of these components has a complex internal structure in terms of composition of other components; in this case study we consider the *NCC* internal structure while we consider *AAL2RIClient* and *NCIClient* as stubbed. *NCC* is a connections handler providing connectivity services for establishment/release of communication paths between pairs of

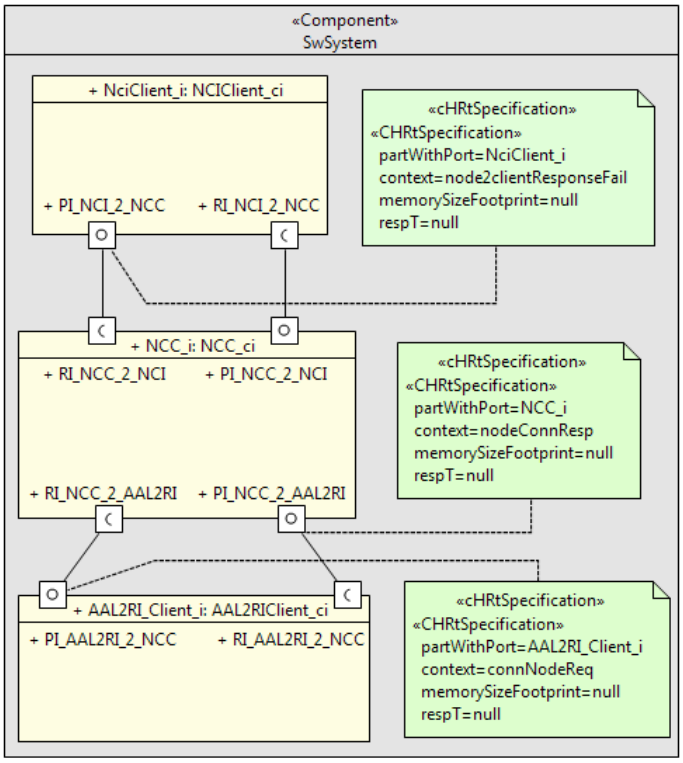

Fig. 3. AAL2 subsystem structural design in CHESS

connection endpoints handled by *AAL2RIClient*. *NCIClient* represents an application asking for services provided by *NCC* and its underlying layers; the components communicate through functional interfaces (function calls) exposed by their provided ports.

The *NCC* component has a complex internal structure (Fig. 4), and in this study we focus on:

- **NodeConnHandler**: which dispatches the incoming connection requests to available NetConn instances;
- **NetConn**: that controls establishment and release of network connections between nodes (NodeConneElem instances);
- **NodeConnElem**: that handles management of network connections within the single node;
- **PortHandler**: which manages connection resources.

Each of these subcomponents has in turn a complex internal structure in term of components composition; in this case-study we consider only the first two levels of decomposition (down to the *NCC*'s internal structure) since we experienced that the number of levels does not critically undermine the approach's

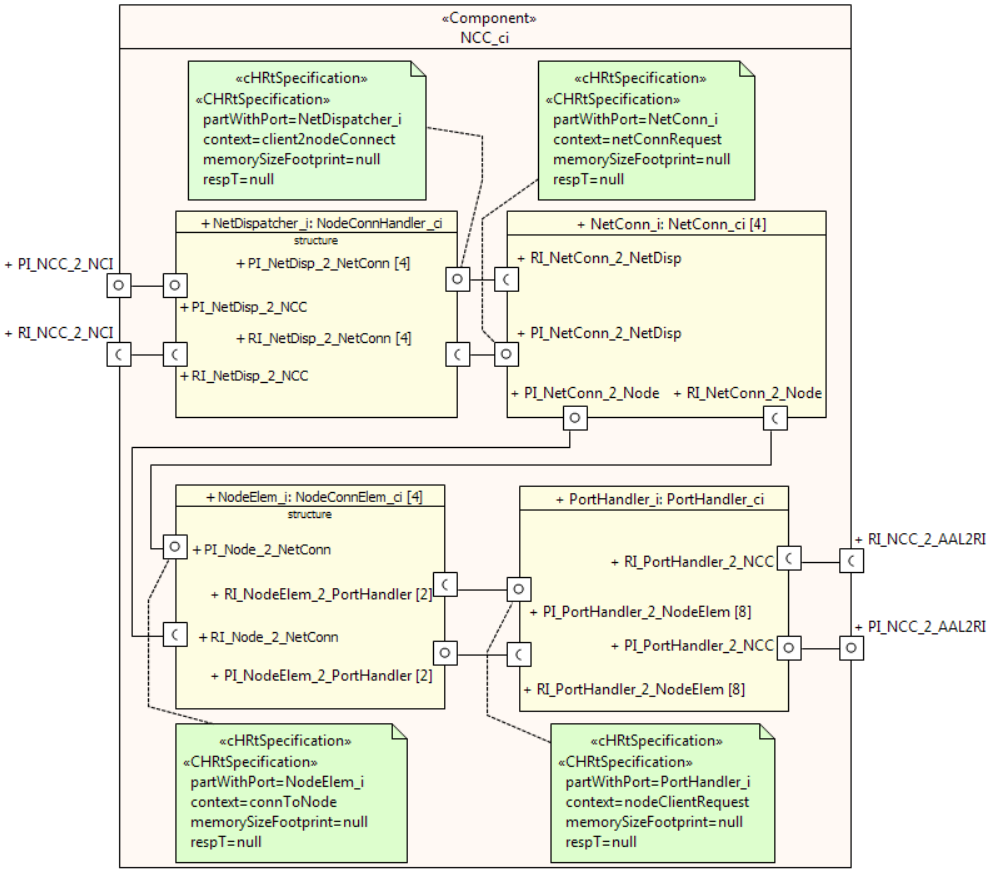

Fig. 4. NCC composite structure in CHESS

scalability. CHESS allows the definition of extra-functional properties by means of decoration of the design models with proper stereotyped annotations. Since in this case-study the properties taken into consideration for monitoring and back-propagation are *execution time* and *allocated memory*, some of the component instances' ports are annotated with the *CHRtSpecification* stereotype, an extension of the MARTE's *RtSpecification* stereotype [21], specifically defined in CHESS for the specification of real-time specific properties. In our case the placeholders used for back-propagation of monitoring results are *respT* (for execution time representation) and *memorySizeFootprint* (for allocated memory representation). The *CHRtSpecification* annotation contains also information regarding the component instance (*partWithPort*) and the specific operation (*context*) in the annotated port to which the real-time specification applies.

In Fig. 3, 4 the decorations on the AAL2 model are depicted and, since the back-propagation has not been performed yet, there is no value specified for the

properties *respT* and *memorySizeFootprint* yet. For readability issues and since a larger number of decorations would not have undermined the validation of the approach, we decided to put one decoration for each component, but there is no actual limitation in such sense. The behavioural definition of the system (*NodeConnHandler* state-machine in Fig. 5) is given by means of state-machines enriched with action code definitions for the involved operations specified by means of the aforementioned ALF. Components are connected by means of ports and links between them. The communication is thereby performed by calling operations on the component's required ports that propagate the invocation to the component owning the provided ports connected to them (note that connected provided and required ports share the same *Interface*).

A typical connection scenario in the AAL2 subsystem is the establishment of a connection between two end-points residing on the same node; this is a constrained case of a more general network-wide connection where the two end-points reside on different nodes and the communication transits through a number of other intermediate nodes in the network.

When *NCIClient* wants to connect two end-points, a connection setup request is sent to *NCC* through the *PI_NCI_2_NCC* interface; such request contains information about the end-points. *NCC* asks for the establishment of a connection segment between the end-points to an external component (not modelled in this case-study) and then sends a request through the *RI_NCC_2_AAL2RI* interface for each end-point to their respective *AAL2RIClient* in order to activate the access to the transport layer for the end-point. Once both end-points have positively responded through their respective *RI_AAL2RI_2_NCC* interface, *NCC* confirms the establishment of the connection to *NCIClient* through the *RI_NCC_2_NCI* interface. In Fig. 5, the state-machine describing the behaviour of *NodeConnHandler* component is shown; ALF code specifying the behaviour of the component's operation *sendResponse* (matching the homonymous state-machine's transition) is also shown in the figure. Communication between components in terms of operations (*node2clientResponseOk(), node2clientResponseFail()*) called on required ports (*RI_NetDisp_2_NCC*) is depicted in the action code fragment.

5 Code Generation Process from Source Models

In this section we demonstrate how the full proposed approach has been validated against the AAL2 subsystem in terms of generation of executable *C++* code and explicit traceability links for back-propagation. In order to be able to perform back-propagation operations, complete traces from source model to related generated execution entity/ies (i.e. code) and monitoring results have to be maintained. Explicit trace links between model elements and code are created by defining a set of ad-hoc transformation rules within the model transformations process responsible for the code generation. Such rules create a mapping between model elements to their respective generated execution units or code blocks by creating and populating a back-propagation model according to the constraints defined in the back-propagation meta-model. Hence, the code generation process

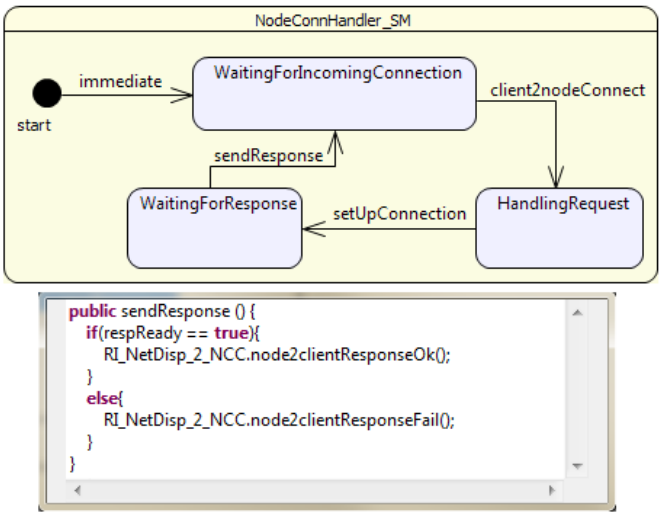

Fig. 5. NodeConnHandler state-machine in CHESS

gives two artefacts as output: (i) generated target code and (ii) back-propagation model containing trace links between source model and produced code.

5.1 Executable Code Generation

The code generation process is composed by a set of model transformations (e.g. model-to-model, model-to-text) that, through iterative and progressive transformation of the input into intermediate representations, generates C++ implementation code from the CHESS models. Given the source models, our solution aims at providing a *full* code generation that entails both static and behavioural description of the system. In this work we provide a high level view of the transformation process (Fig. 6) from design models to the corresponding C++ code specifically enhanced for enabling back-propagation activities and facilitating controlled code injections for analysis purposes; a detailed description of such process will be part of a separate upcoming publication. The code generation process relies on models and transformations as main artefacts for the generation of target code and is composed by the following tasks:

1. A set of model-to-model transformations, defined using the Operational QVT (QVTo)[4] transformation language, act on the CHESS model to generate two different intermediate artefacts:

[4] The QVT language, abbreviation of Query/View/Transformation, is defined [36] by the OMG, Object Management Group. The name of the language recalls its three-parts language that allows the description of queries to get a selection of model elements, the definition of restricted views of a model for cutting away aspects of the model not relevant to a user or domain, and transformations between models, respectively

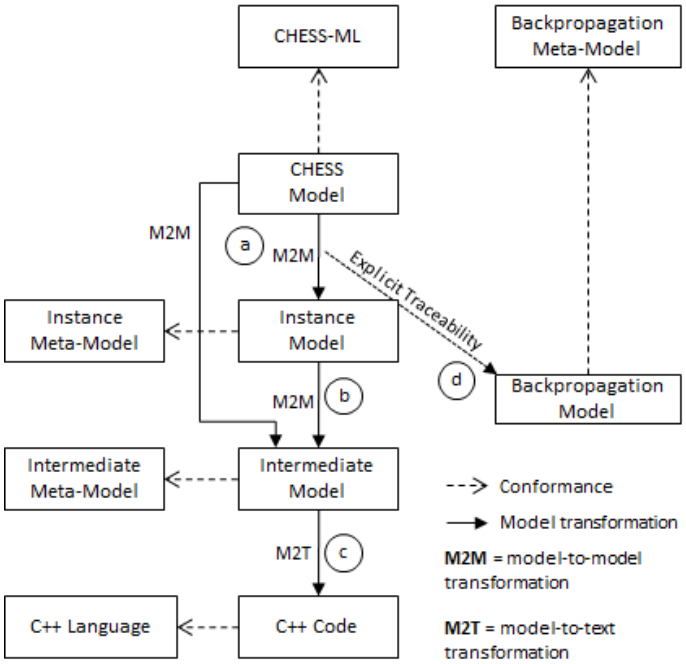

Fig. 6. Code Generation Process

- Instance model: that represents unwound components and ports instances according to their multiplicities for enabling a correct generation of the communication links between components instances at code level;
- Intermediate model: that represents the main intermediate artefact in the process and contains all the needed information, both structural and behavioural, derived from the source model to generate full implementation code (Fig. 6a).

Each of these models conform to its corresponding meta-model that we expressly defined for the code generation process. Moreover, during this task, explicit traceability links are created (Fig.6d); the description of such step is given in the next section (5.2);

2. The intermediate model is enriched with other information deriving from both the instance model and the CHESS model by means of in-place [3] model-to-model transformations (Fig. 6b);
3. Finally the C++ implementation code is generated through model-to-text transformations, defined using the statically-typed template *Xpand* language [37], taking as input the sole intermediate model (Fig. 6c).

5.2 Traceability Links and Back-propagation Model

The result of code generation activities is a set of C++ code files and the portion of the back-propagation model containing traceability information between mod-

elling elements (i.e. components, ports, operations and extra-functional properties) and code execution units in terms of the code artefacts implementing the system functionalities. The back-propagation model is created during the code generation process through model-to-model transformations defined by means of the QVTo transformation language, more specifically using the Eclipse implementation [38].

As aforementioned, the first step of the code generation process consists of a model-to-model transformation acting on the CHESS model to generate the instance model, that represents unwound components and ports instances according to their multiplicities. During this task the back-propagation model is generated too. The transformation code portion depicted in Fig. 7 represents the creation of the trace elements related to each of the instantiated component instances. Navigating the CHESS model from the root component through all its composition levels, for each component instance present in the model a number of trace elements are created to trace the properties monitored at operation level following this steps:

1. Lines 66-87: for each subcomponent directly contained by the current component, a *ModelElementInstance* in the back-propagation model is created. Particularly important in this process is that the containing component is set as *parent* of the *ModelElementInstance* in order to maintain the containing hierarchy crucial for back-propagation activities. In fact, it may happen that different instances of the same component type are defined in different parts of the model with ambiguous identity; by maintaining the containment relationships from the root component we are able to univocally identify the different instances of a same component type and correctly perform back-propagation. Moreover, since at code level the different component instances are identified by a progressive unique numerical identifier assigned during the code generation, the model element instance will also need to inherit such information in order to allow correct injection of the monitoring results to the right placeholder in the back-propagation model
2. Lines 92-105: for each operation defined in the subcomponent, a corresponding *FunctionalUnit* is created in the back-propagation model together with the *FunctionalUnitProperty* elements the properties that are meant to be monitored: *respT* and *memorySizeFootprint*, respectively representing execution time and memory allocation in the *CHRtSpecification* stereotyped annotation. Due to its relatively lower complexity, component level granularity properties have been left aside in this case-study but still present in the actual implementation. In fact, the generation of such properties is performed in the same way as the ones shown, but *ModelElementProperty* elements would be created instead of *FunctionalUnitProperty* and they would be associated to *ModelElementInstance* elements instead of *FunctionalUnit*
3. Lines 106-110: for each operation, a corresponding C++ code function will be created and its execution will be used for monitoring and compute values for the defined properties. An *ExecutableUnit* element has to be created for this purpose, and the previously defined *FunctionalUnitProperty* elements

```
// create instances for components and ports
mapping UML::Component::comp2instances(in index : Integer, in rel_ind : Integer,
                    in instanceN : String, in mul : Integer,
            inout baModel : BACKPROPAGATION::BackpropagationModel,
    inout actualBack : BACKPROPAGATION::ModelElementInstance) : INSTANCE::Component{
 // create component instance
 name := self.name;
 id := index;
 rel_id := rel_ind;
 instantiatable := true;
 istanceName := instanceN;
 mult := mul;

 // for each subcomponent proceed with back-propatation information creation
 // and instantiation
 self.ownedAttribute->forEach(a){
  if(a.type.oclIsTypeOf(UML::Component))then{
    var i:=1;
    while(i<=a.upperBound()){
     var ind := inst.objectsOfType(INSTANCE::Component)->size()+1;

     // instantiation of the trace elements related to the component instances
     // and their operations
     var temp := a->map comp2back(ind,baModel)->asOrderedSet()->first();

     // set the current component as parent of the subcomponent in the
     // back-progation model for further back-propagation activities
     temp.parent:=actualBack;

     // for each operation defined for the subcomponent create a related trace
     // element considering respT and memorySizeFootprint of the stereotype
     // CHRtSpecification as monitored properties
     if(a.type.oclAsType(UML::Component).ownedOperation->asOrderedSet()->size()>0)then{
      a.type.oclAsType(UML::Component).ownedOperation->asOrderedSet()->forEach(t){
       var func := object BACKPROPAGATION::FunctionalUnit{name:=t.name};
       var propT := object BACKPROPAGATION::FunctionalUnitProperty{
                                category:="CHRtSpecification";
                                    name:="respT";
                                        has:=object BACKPROPAGATION::PropertyValue{}};
       var propM := object BACKPROPAGATION::FunctionalUnitProperty{
                                category:="CHRtSpecification";
                                    name:="memorySizeFootprint";
                                        has:=object BACKPROPAGATION::PropertyValue{}};
       func.has += propT;
       func.has += propM;
       temp.hasOp+=func;
       var targetF := object BACKPROPAGATION::ExecutableUnit{
                                name:=a.type.oclAsType(UML::Component).name+
                                    "_"+t.name; isPartOf:=baModel.tracesEE->first()};
       targetF.monitors+=propT;
       targetF.monitors+=propM;
       baModel.tracesElem+=object BACKPROPAGATION::TraceElement{
                        id:=name:=t.name+"__2__"+
                            a.type.oclAsType(UML::Component).name+"_"+
                                                t.name+"__"+propT.name;
                            modelSource := temp; sourceFunction:=func;
                                targetFunction:=targetF; _property:=propT};
       baModel.tracesElem+=object BACKPROPAGATION::TraceElement{
                        id:=name:=t.name+"__2__"+
                            a.type.oclAsType(UML::Component).name+"_"+
                                                t.name+"__"+propM.name;
                            modelSource := temp; sourceFunction:=func;
                                targetFunction:=targetF; _property:=propM};
       };
      }endif;
```

Fig. 7. Creation of Trace Elements

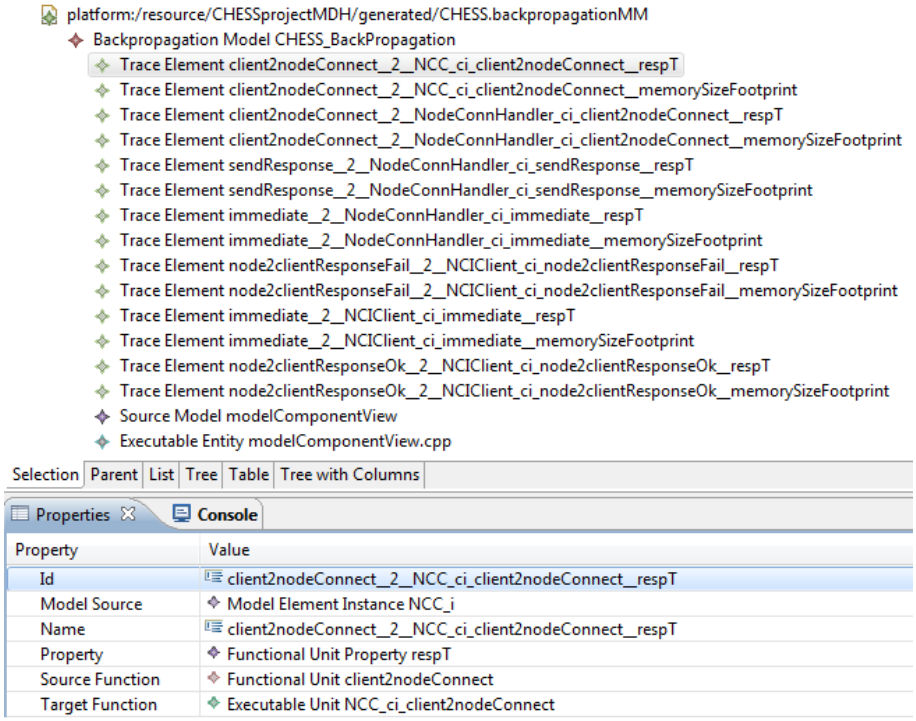

Fig. 8. Trace Element details in the AAL2's Back-propagation Model

are linked to it in order to complete the traceability chain (model-code-properties)

4. Lines 111-122: finally a *TraceElement* is created for each of the properties to be monitored according to the definition given in section 3.1. In Fig. 8 the details of one of the trace elements created during the code generation process for the AAL2 subsystem is depicted. The meaning of such trace can be summarized as follows: the trace element *client2NodeConnect_2_NCC_ci_client2NodeConnect_respT* represents the trace link between the *client2nodeConnect* operation, and the monitored property *respT*, defined for the component instance *NNC_ci* and the code function *NCC_ci_client2nodeConnect*.

In Fig. 9 the portion of the back-propagation model that we consider for the next back-propagation steps is depicted.

6 Code Execution and Back-propagation to Source Models

In this section our running case-study is used to show the conclusive phases of the approach in terms of: (i) code execution monitoring activities, (ii) management

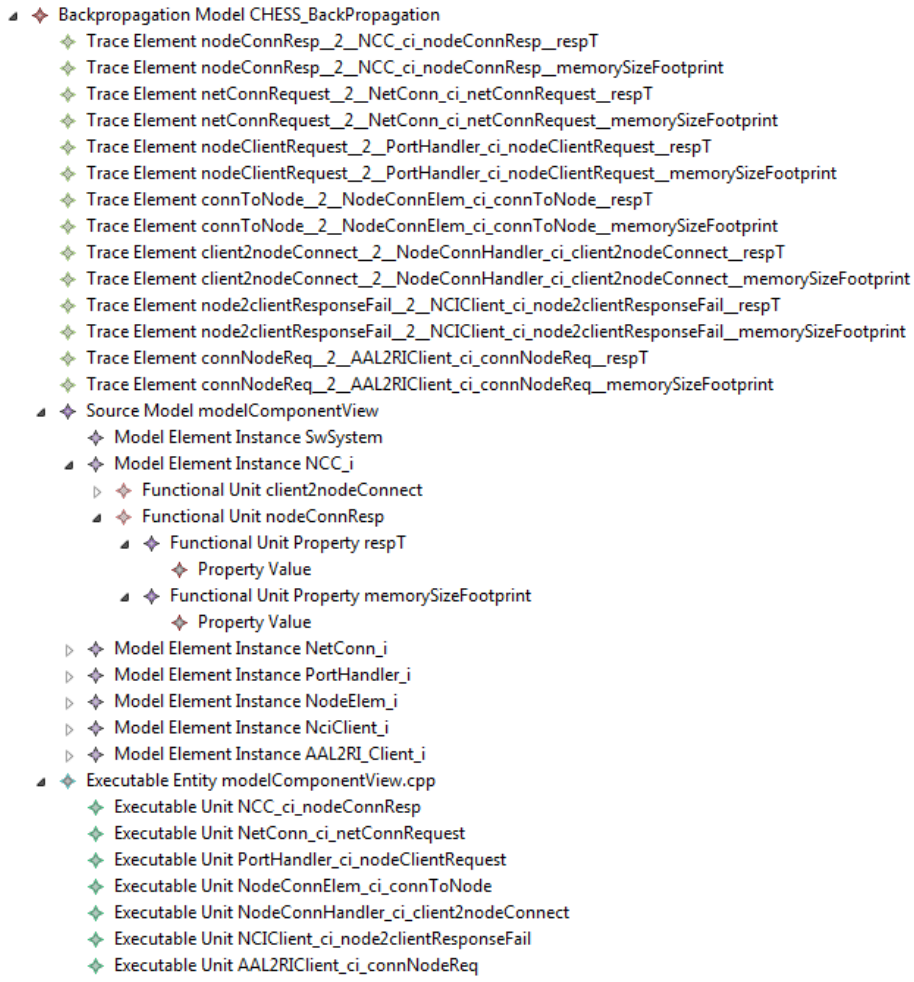

Fig. 9. Portion of the generated Back-propagation Model related to the AAL2 Subsystem

of monitoring results and (iii) propagation of such results back to the source model in CHESS.

6.1 Monitoring Results Management

In the same way as the model transformation responsible for the code generation creates the trace links between source model and code in terms of trace elements in the back-propagation model, the information about extra-functional properties values calculated during monitoring activities has to be injected in the back-propagation model in order to have the complete chain model-code-values needed for propagating those values back to the source model. The way to perform such injection depends on both code generation and monitoring output format; it could in fact vary from model-to-model to text-to-model transformation from monitoring results to the back-propagation model. This can be considered a variable point of the round-trip support approach in the sense that it is hard to generalize for a multitude of different tools, but rather adapted to the output of each of them if such output can not be adapted to the expected input (text file with a specific format) of the proposed injecting transformation.

In this work we implement such injection by means of text-to-model transformation since the monitoring activities give a textual description of the computations as output. More specifically, the monitoring produces a text file which is a set of four-token lines formatted as follows: *ExecutableUnit ModelElementInstance.id Property Value*. The transformation itself is implemented in Java and, taking as input the back-propagation model *BM* and a monitoring output file *MF*, acts as follows:

```
for each line l in MF do
  for each token t in l do
    if n == 1 then
      execUnit = t;
    else
      if n == 2 then
        id = t;
      end if
    else
      if n == 3 then
        property = t;
      end if
    else
      if n == 4 then
        value = t;
      end if
    end if
  end for
  BMtrace = BM.search(execUnit, id, property);
  if BMtrace! = NULL then
    BMtrace.property.value = value;
```

end if
end for

MF is navigated line by line, and each of which is tokenized according to the defined format; the tokens *ExecutableUnit*, *ModelElementInstance.id* and *Property* represent the information for which a match has to be sought in *BM*. The identifier (*ModelElementInstance.id*) related to the *ModelElementInstance* is crucial for identifying from which component instance in the code the property value is derived and thereby to which trace element has to be injected in the back-propagation model. Once a match is found, which means that there is a trace element *BMtrace* linking *Property* with *ExecutableUnit* and *ModelElementInstance* in the back-propagation model, then the token *Value* is injected into the correct placeholder pointed by *Property* in the trace element *BMtrace* to add the calculated value to the related property.

6.2 Performing the Back-propagation

Once the back-propagation model has been completed with property values derived from the monitoring activities, the final step of the process, that is to say propagating such values back to the source model by injection, can be performed. This task is crucial for enabling models optimization and produce correct-by-construction system implementation.

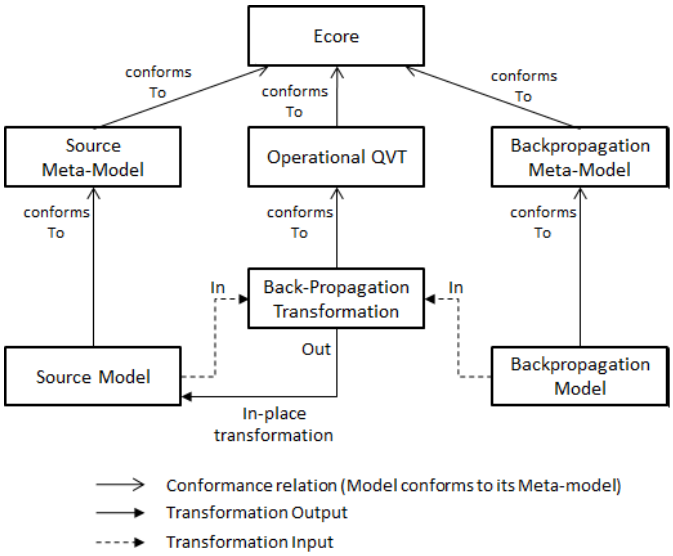

Fig. 10. Back-propagation Transformation

Since the information to be back-propagated to the source model is in turn stored in a model (i.e. the back-propagation model), the injection is performed

through a QVTo model-to-model transformation that, taking as input the source model and the back-propagation model, operates a set of in-place transformations on the source model by enriching it with the property values stored in the back-propagation model (Fig. 10).

A back-propagation model is composed by a non-empty set of trace elements *TE* defined as quadruples <*ME, FU, EU, FUP*> where *ME* is a model element contained in a source model *SM*, *FU* represents an operation/method defined in *ME* and *FUP* represents an extra-functional property defined for *FU* and meant to be calculated by monitoring the execution of the executable unit *EU*.

The transformation (Fig. 11) takes as input the back-propagation model *BM* and source model *SM*, as well as the meta-models to which they conform to, and gives as output an enriched version of *SM* achieved by means of in-place transformation (i.e. SM is both input and output of the transformation) acting as described in the following pseudo-code:

for *each trace element* $TE = (ME, FUP)$ *in* BM **do**
  $SMproperty = SM.search(ME, FUP)$;
  if $SMpropertyexists$ **then**
    $SMproperty.value = P.value$;
  end if
end for

BM is navigated and for each trace element *TE* a match is sought in *SM*; if model element *ME* and property *FUP* traced by *TE* match with a corresponding pair in *SM* then the value associated to *FUP* in *TE* is injected into the matching property in *SM*.

The code produced is C++ and the Linux API *getrusage* [39] is used for getting monitoring information from its execution. The execution of the generated C++ code produces a result text file according to the format described in section 6.1. In our example information regarding monitored properties for the AAL2 subsystem's components are stored in the monitoring results file, after code execution, as follows:

NCIClient_ci_node2clientResponseFail 13 respT 1201
NCIClient_ci_node2clientResponseFail 13 memorySizeFootprint 8716
NCC_ci_nodeConnResp 2 respT 3402
NCC_ci_nodeConnResp 2 memorySizeFootprint 12327
NodeConnHandler_ci_client2nodeConnect 12 respT 6004
NodeConnHandler_ci_client2nodeConnect 12 memorySizeFootprint 4550
NetConn_ci_netConnRequest 3 respT 1457
NetConn_ci_netConnRequest 3 memorySizeFootprint 6093
PortHandler_ci_nodeClientRequest 7 respT 3990
PortHandler_ci_nodeClientRequest 7 memorySizeFootprint 8770
AAL2RIClient_ci_connNodeReq 14 respT 1805
AAL2RIClient_ci_connNodeReq 14 memorySizeFootprint 9982

```
modeltype CHESS uses "http:///schemas/CHESS/_PfAJsMe6Ed-7etIj5eTw0Q/19";
modeltype UML uses "http://www.eclipse.org/uml2/3.0.0/UML";
modeltype MARTE uses "http://www.eclipse.org/papyrus/MARTE/1";
modeltype ANNOTATION uses "http://se.mdh.chess.backpropagationMM";

transformation backPropCHESS(in backModel : ANNOTATION, inout chessModel : CHESS);

// definifition of the CHESS extra-functional properties taken into consideration
property rtspecification = "CHRtSpecification";
property respT = "respT";
property memory = "memorySizeFootprint";

// for each trace element in the back-propagation model try to propagate to CHESS model
main() {
    backModel.rootObjects()->asOrderedSet()->
                    first()[ANNOTATION::BackpropagationModel].tracesElem->forEach(te){
        te.propagate();
    };
}

// main propagation function: look for the element in the CHESS model, and if found
// update the property value
helper ANNOTATION::TraceElement::propagate(){
    var compView := chessModel.objectsOfType(CHESS::Core::CHESSViews::ComponentView)->
        asOrderedSet()->first().oclAsType(CHESS::Core::CHESSViews::ComponentView);
    var res := self.modelSource.findElement(compView);
    if(not res.oclIsInvalid())then{
        res.child.updateValue(self._property, self.sourceFunction, res.parent);
    }endif;
    return;
}

// given the information in the trace element (ModelElementInstance, Property,
// FunctionalUnit) find the right component instance in the CHESS model for
// back-propagation
helper ANNOTATION::ModelElementInstance::findElement(
                                in compView : CHESS::Core::CHESSViews::ComponentView)
                            : child : UML::Property, parent : UML::Component{
    parent := compView.base_Package.allOwnedElements()->
                                select(oclIsKindOf(UML::Component))[UML::Component]->
                            select(name = self.parent.type)->asOrderedSet()->first();
    child := parent.attribute->select(name = self.name and type.name = self.type)->
                                asOrderedSet()->first().oclAsType(UML::Property);
}

// once the component instance is found, the values are propagated back to the
// apposite placholder(s)
helper UML::Property::updateValue(in prop : ANNOTATION::Property,
                                in func : ANNOTATION::FunctionalUnit,
                            in parent : UML::Component){
    parent.ownedComment->forEach(c){
        if(prop.category=rtspecification)then{
            var stereotype := c.getStereotypeApplications()->
                select(metaClassName()=prop.category)->asOrderedSet()->
                    first().oclAsType(CHESS::RTComponentModel::CHRtSpecification);
            if(stereotype.partWithPort.name = self.name and
                        stereotype.context.name = func.name)then{
                switch{
                    case(prop.name=respT)
                                    stereotype.respT:=prop.has.value;
                    case(prop.name=memory)
                                    stereotype.memorySizeFootprint:=prop.has.value;
                };
            }endif;
        }endif;
    };
}
```

Fig. 11. Model-to-Model QVTo Transformation for Back-propagation

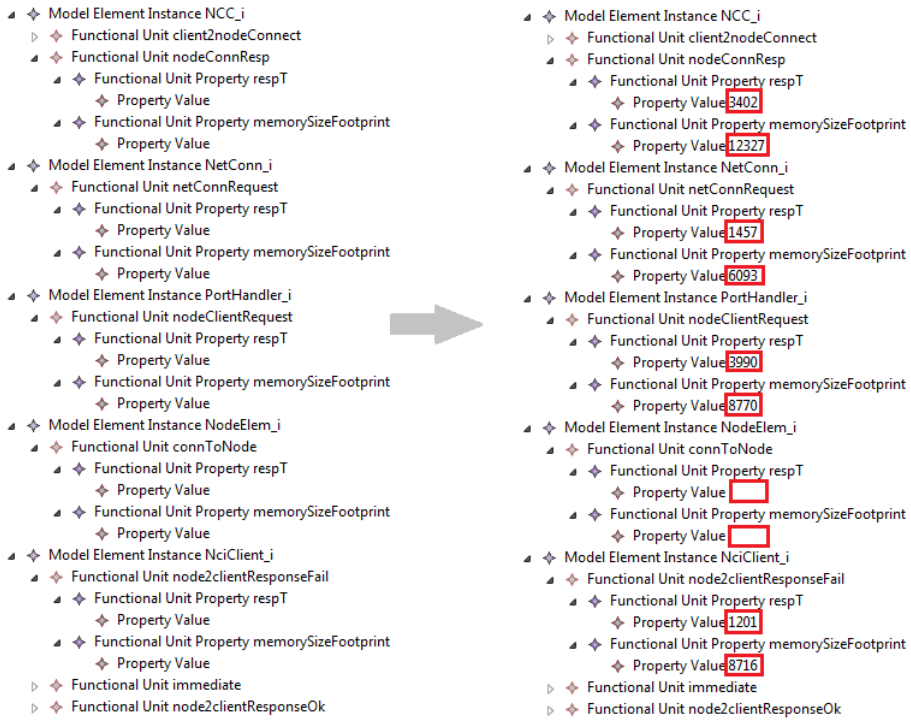

Fig. 12. Injection of the Monitoring results to the Back-propagation Model

Once the execution is completed, the results in terms of monitored properties values are injected into the back-propagation model through the Java transformation described in section 6.1 as shown in Fig. 12; note that, since during monitoring activities no values were gathered for the properties defined for the *NodeElem_i* component instance, no value is injected into the back-propagation model. At this point all the information to be propagated back to the AAL2 subsystem model is stored in the back-propagation model. In order to finally complete the round-trip path, this information is to be put in the apposite AAL2 subsystem model elements' placeholders through the QVTo back-propagation transformation, described in section 6.2, fed with the AAL2 model defined in CHESS and the complete back-propagation model as input artefacts. The transformation operates a set of in-place transformations on the AAL2 model giving as output an enriched version of the same model; the properties monitored during the code execution are now equipped with the related monitored values. In Fig. 13, 14 the AAL2 subsystem and its *NCC* composite component with back-propagated values for execution time (*respT*) and allocated memory (*memorySizeFootprint*) are shown. The round-trip process has produced an extra-functionally enriched version of the source model and it is now possible for the developing team to validate the system and finally perform further op-

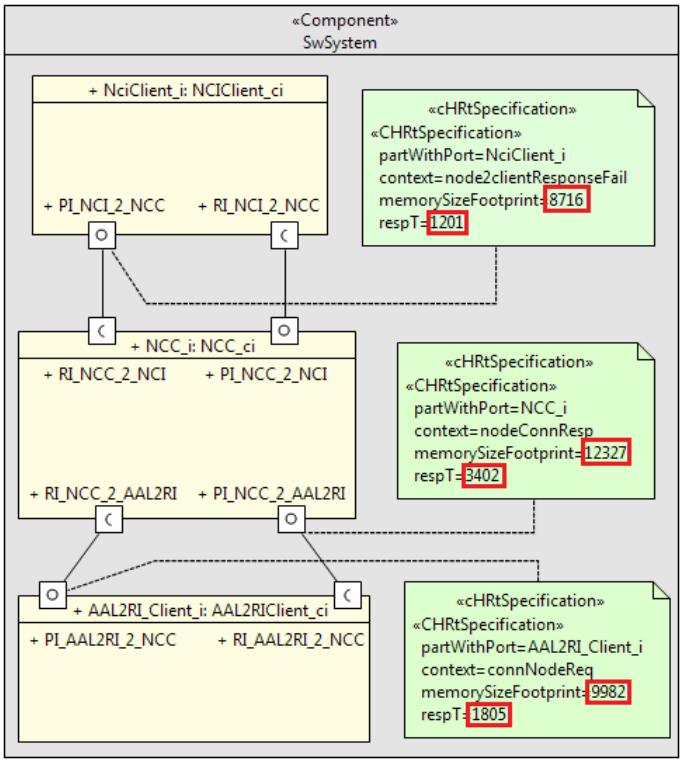

Fig. 13. Back-propagation to AAL2 subsystem in CHESS

timization activities directly at modelling level rather than at code level with a consequent conservation in terms of consistency among the artefacts and their properties at each considered abstraction level.

7 Discussion

Due to the multi-step nature of the proposed approach, questions may arise regarding consistency issues among the involved artefacts, both vertically and horizontally, from source models to generated code and vice versa. In fact, allowing human interference at any of the described steps may undermine the validity of the entire process at two levels:

– Code generation: breaking the consistency among artefacts during the code generation voids the final aim of generating correct-by-construction system implementation from design models. That is the reason for which the code generation, including all the intermediate steps (i.e. trace links creation), is an atomic process kept transparent to the developer who is not able to modify any of the intermediate artefacts;

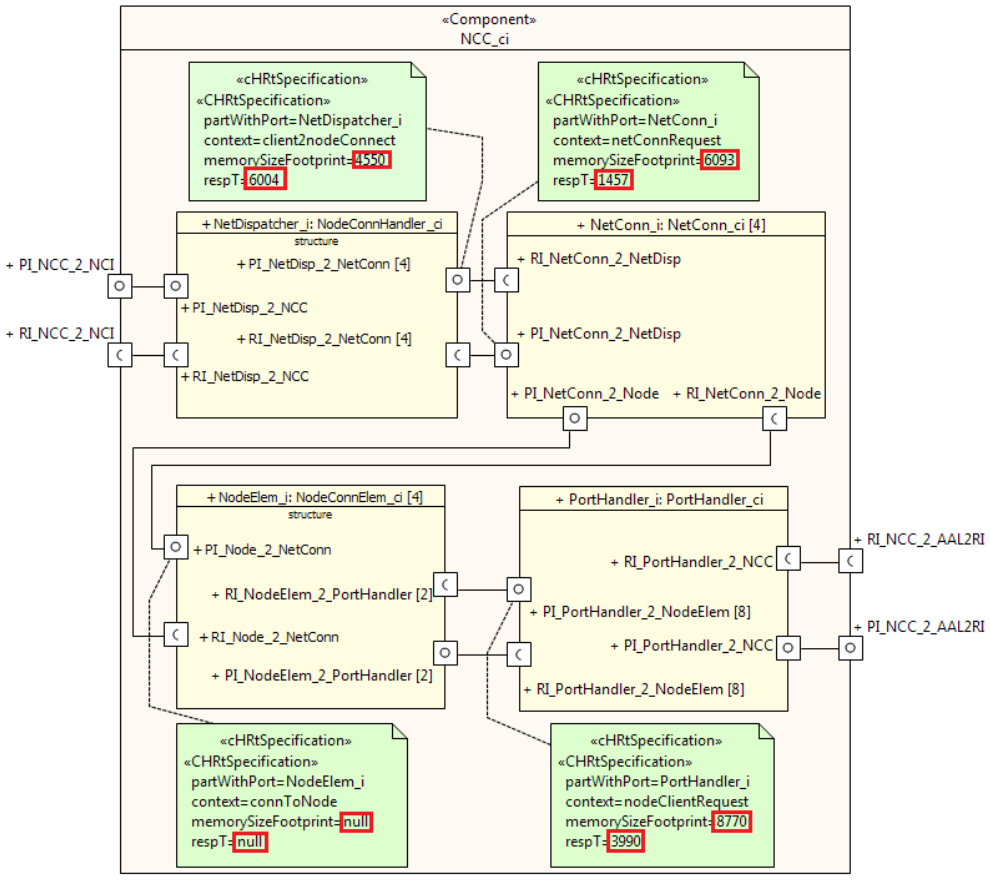

Fig. 14. Back-propagation to the NCC composite component in CHESS

– Back-propagation: the user has control only over source models and generated code. Modifying such artefacts, the developer may cause inconsistencies during the back-propagation phase and hence jeopardize the reliability of the back-propagated values in relation to the system models to which they are back-propagated and the code they generated from. That is the reason for which, in order for the proposed approach to guarantee that gathered information is correctly and consistently back-propagated to the source models, generated code is not meant to be manually edited.

Validating the approach against a complete and extensive case-study and on top of a different framework gave us the possibility to make comparisons with the preliminary validation described in [34] and evaluate important characteristics such as scalability and reusability of the proposed solution.

Concerning scalability, we analysed the behaviour of the approach from the perspective of the entire process as well as stepwise. Moreover, within the same

case-study we tested several source model sizes in order to thoroughly evaluate possible consequences in terms of scalability. From a process-wise perspective, the proposed solution resulted very scalable up to 10^4 component instances hence degrading going toward 10^5; in any case the process always accomplished its goals. Analysing this result from a step-wise perspective, we noticed that the least scalable tasks were those responsible for the code generation; the reason is quite straightforward and relies on the computation's complexity of the involved transformations. On the other hand, the better scalability of back-propagation tasks (e.g. monitoring results management and actual back-propagation to source model) resulted to be less dependent of the source model's size; the scalability of these tasks may in any case vary if the approach were applied to other modelling languages. Intermediate artefacts' size may grow proportionally to source model's; the fact that they are meant to be transparent and handled only by the process itself relieves the developer of the burden of understanding and managing them and lowers possible overheads deriving from their textual and graphical rendering.

Applying the approach to a different case-study modelled with a different modelling language and on top of a different framework, the reusability capabilities of the approach have been challenged and evaluated. Code generators have been specifically implemented for the CHESS framework, while the other steps of the process adapted to correctly deal with the CHESS environment and modelling language. The overhead of such adaptations can vary depending of the chosen modelling language, target implementation code, and so on. Anyhow, if considering the core generic artefact of the process, namely the back-propagation meta-model, and the tasks performed on it, the adaptation overhead has been much lower than expected. Moreover, driven by the possibilities given by the CHESS framework in defining extra-functional properties, we enhanced the back-propagation meta-model in order to enable traceability as well as back-propagation of properties defined at different levels of granularity; such enhancement makes traceability and back-propagation capabilities more powerful and easily adaptable to a wider set of different modelling frameworks and needs.

8 Conclusion

In this work we lay foundations and motivations for a round-trip engineering approach which aims at supporting a model-driven and component-based development process cycle in ensuring extra-functional properties preservation from models to generated implementation code. Such capability, achieved through generation of 100% of the implementation code and corresponding traceability together with back-propagation from code to model level makes the contribution of this paper unique.

The approach is meant to assert the ability of the development process in providing composition with guarantees in terms of preserved extra-functional requirements at generated code level and thus testify its goodness and its capabilities in driving and helping the user in the development of extra-functionally

correct-by-construction real-time embedded systems. Challenges and issues related to each single step of the approach have been highlighted and related solutions have been proposed and tested on a selected case-study.

After a preliminary validation against a system designed on top of the ProCom platform, the round-trip support has been enriched with its own C++ code and trace links generators and validated at a larger scale against a more complete and extensive case-study on top of a different framework, CHESS. From the design models, created using the CHESS modelling language and following the CHESS methodology, C++ code and trace links in terms of a back-propagation model are automatically generated. The execution of the implementation code is then monitored and selected extra-functional properties (e.g. execution time and allocated memory) measured at operation level; apposite transformations are defined to perform injection of such values into the back-propagation model. Such model is then taken as input by an apposite model-to-model transformation for propagating the computed property values back into the related model elements' placeholders at modelling level. Once an iteration of the process is completed, the developers have at their proposal a set of enriched source models from which it is possible to extract precious information about the modelled system in terms of monitored quality attributes. Finally the models can be optimized and the process re-iterated until generated implementation code achieves desired preservation of selected extra-functional properties.

9 Future Work

Future research directions will encompass the extension of the proposed approach by taking into account management and evaluation of properties like safety and security, which usually differ from properties measurable by means of computed values.

Given the ability of the approach of supporting different levels of granularity both for traceability and back-propagation, a possible enhancement would be the possibility for the developer to select the wished level of granularity at modelling level at the beginning of a process iteration; this would make the process able to calculate only the desired traceability information reducing the needed computation efforts, for instance, when generating the back-propagation model. Moreover, a further enhancement would be the ability for the code and trace generation process to create the trace links taking into consideration the sole extra-functionally decorated model elements; again in this case the computation efforts needed for back-propagation activities and management of related artefacts (e.g. back-propagation model) could be firmly reduced.

Acknowledgements

The research work presented in this paper has been supported by the CHESS project (Composition with Guarantees for High-integrity Embedded Software Components Assembly, ARTEMIS Joint Undertaking grant nr. 216682) [7].

References

1. Bézivin, J.: On the unification power of models. Software and System Modeling **4** (2005) 171–188
2. Kent, S.: Model driven engineering. In: Proceedings of the Third International Conference on Integrated Formal Methods. IFM '02, London, UK, UK, Springer-Verlag (2002) 286–298
3. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture. (2003)
4. OMG: Model Driven Architecture. `http://www.omg.org/cgi-bin/doc?omg/00-11-05.pdf` (2000)
5. Cancila, D., Passerone, R., Vardanega, T., Panunzio, M.: Toward correctness in the specification and handling of non-functional attributes of high-integrity real-time embedded systems. Industrial Informatics, IEEE Transactions on **6** (2010) 181–194
6. Chodrow, S.E., Jahanian, F., Donner, M.: Monitoring and debugging of distributed real-time systems. IEEE Computer Society Press, Los Alamitos, CA, USA (1995) 103–112
7. ARTEMIS-JU-216682: CHESS. `http://chess-project.ning.com/` (2009)
8. Handel, R., Huber, M.N., Schroder, S.: ATM networks: concepts, protocols, applications. 2nd ed. edn. Addison-Wesley Pub. Co., Wokingham, England ; Reading, Mass. (1994)
9. Crnkovic, I., Larsson, M.: Building Reliable Component-Based Software Systems. Artech House, Inc. (2002)
10. Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., Neema, S.: Developing Applications Using Model-Driven Design Environments. Computer **39** (2006)
11. Childs, A., Greenwald, J., Jung, G., Hoosier, M., Hatcliff, J.: Calm and cadena: metamodeling for component-based product-line development. Computer **39** (2006) 42 – 50
12. IBM: Rational Software Architect 8.0. `http://www.ibm.com` (2010) [Online. Last access: 23/05/2011].
13. SINTEF-ITC: Component and Model-Based Development Technology: COMET. `http://www.modelbased.net/comet/1c\_Tools\_html.html` (2006) [Online. Last access: 23/05/2011].
14. Vardanega, T.: Property preservation and composition with guarantees: From assert to chess. In: Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on. (2009) 125 –132
15. Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: 2007 Future of Software Engineering. FOSE '07, Washington, DC, USA, IEEE Computer Society (2007) 171–187
16. Brosch, F., Koziolek, H., Buhnova, B., Reussner, R.: Parameterized reliability prediction for component-based software architectures. In: QoSA. (2010) 36–51
17. Crnkovic, I.: Component-based software engineering for embedded systems. In: Proceedings of ICSE'05, (ACM) 712–713
18. Ko, J.M., Kim, C.O., Kwon, I.H.: Quality-of-service oriented web service composition algorithm and planning architecture. Journal of Systems and Software **81** (2008) 2079–2090

19. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Proceedings of the 12th international conference on World Wide Web. WWW '03, New York, NY, USA, ACM (2003) 411–421
20. OMG: UML Profile for Modeling QoS, Fault Tolerance Characteristics and Mechanisms Specification. http://www.omg.org/spec/QFTP/1.1/PDF/ (2008)
21. OMG: UML Profile For MARTE: Modeling And Analysis Of Real-Time Embedded Systems, v1.1. http://www.omg.org/spec/MARTE (2011)
22. ESA: ASSERT Project. http://www.assert-project.net/Assert-project (2007)
23. Navabi, Z., Day, S., Massoumi, M. (In: Proceedings of VHDL International User Forum) 185–195
24. Mahadevan, G., Armstrong, J.R.: Automatic Back Annotation of Timing into VHDL Behavioral Models. In: Proceedings of VHDL International User Forum. (1995) 27–41
25. Hegedüs, Á., Bergmann, G., Ráth, I., Varró, D.: Back-annotation of simulation traces with change-driven model transformations. In: Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on. (2010) 145 –155
26. Guerra, E., Sanz, D., Díaz, P., Aedo, I.: A transformation-driven approach to the verification of security policies in web designs. In: Proceedings of the 7th international conference on Web engineering. ICWE'07, Berlin, Heidelberg, Springer-Verlag (2007) 269–284
27. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Syst. J. **45** (2006) 515–526
28. Galvao, I., Goknil, A.: Survey of traceability approaches in model-driven engineering. In: Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International. (2007) 313
29. Watterson, C., Heffernan, D.: Runtime verification and monitoring of embedded systems. Software, IET **1** (2007) 172–179
30. Arora, D., Ravi, S., Raghunathan, A., Jha, N.K.: Secure embedded processing through hardware-assisted run-time monitoring. (2005) 178 – 183 Vol. 1
31. Grammel, B., Kastenholz, S.: A generic traceability framework for facet-based traceability data extraction in model-driven software development. In: Proceedings of the 6th ECMFA Traceability Workshop. ECMFA-TW '10, New York, NY, USA, ACM (2010) 7–14
32. Olsen, G.K., Oldevik, J.: Scenarios of traceability in model to text transformations. In: Proceedings of the 3rd European conference on Model driven architecture-foundations and applications. ECMDA-FA'07, Berlin, Heidelberg, Springer-Verlag (2007) 144–156
33. Müller, H.A., Jahnke, J.H., Smith, D.B., Storey, M.A., Tilley, S.R., Wong, K.: Reverse engineering: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering. ICSE '00, New York, NY, USA, ACM (2000) 47–60
34. Ciccozzi, F., Cicchetti, A., Sjödin, M.: Towards a round-trip support for model-driven engineering of embedded systems. In: EUROMICRO-SEAA. (2011) 200–208
35. OMG: Action Language For FoundationalUML - ALF. http://www.omg.org/spec/ALF/ (2010)
36. OMG: Query/View/Transformation, V1.0. http://www.omg.org/spec/QVT/1.0/PDF/ (2008)
37. Eclipse Projects: Xpand. http://www.eclipse.org/modeling/m2t/?project=xpand (2011)

38. Boyko, S., Dvorak, R., Igdalov, A.: The Art of Model Transformation with Operational QVT. http://www.eclipse.org/m2m/qvto/doc/EclipseCon_2009.ppt (2009)
39. Linux Die: Linux 2.6.9 Manual. http://linux.die.net/man/2/getrusage (2006)