Round-trip support for extra-functional property management in model-driven engineering of embedded systems

Federico Ciccozzi *, Antonio Cicchetti, Mikael Sjödin

School of Innovation, Design and Engineering (IDT), MRTC, Mälardalen University, Västerås, Sweden

Abstract

Context: In order for model-driven engineering to succeed, automated code generation from models through model transformations has to guarantee that extra-functional properties specified at design level are preserved at code level.

Objective: The goal of this research work is to provide a full round-trip engineering approach in order to evaluate quality attributes of the embedded system by code execution monitoring as well as code static analysis and then provide back-propagation of the resulting values to modelling level. In this way, properties that can only be roughly estimated statically are evaluated against observed values and this consequently allows to refine the design models for ensuring preservation of analysed extra-functional properties at code level.

Method: Following the model-driven engineering vision, (meta-) models and transformations are used as main artefacts for the realisation of the round-trip support which is finally validated against an industrial case study.

Result: This article 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 observed values to modelling level. The technique is validated against an industrial case study in the telecommunications applicative domain.

Conclusion: Preservation of extra-functional properties through appropriate description, computation and evaluation makes it possible to reduce final product verification and validation effort and costs by generating correct-by-construction code. The proposed round-trip support aids a model-driven component-based development process in ensuring a desired level of extra-functional properties preservation from the source modelling artefacts to the generated code.

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 model aspects in domain-specific terms. In this respect, Model-Driven Engineering (MDE) aims to assist the system development by creating, manipulating and maintaining models that provide abstractions of 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 their 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, this goal is too often seen as the very final step of an MDE approach [4]. On the one hand, preservation of Extra-Functional Properties (EFPs) 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 generating correct-by-construction code. The proposed round-trip support aids a model-driven component-based development process in ensuring a desired level of extra-functional properties preservation from the source modelling artefacts to the generated code.
modelling level (or design level) as the one where design models are defined after proper analysis of requirements specification while code level as the one where activities on code, such as static analysis or monitoring of execution, are performed. Moreover, the results coming from analysis and monitoring activities are addressed as observed values.

In order to be able to perform an evaluation of expected against observed EFPs, the model-code abstraction gap must be bridged; this can be performed by propagating the EFPs’ observed values back to the design models. Hence, we hereby claim that the generation of code and its execution on target platforms should be rather seen as a possibly transitional step in the development; the results coming from the execution would be utilised as an enrichment of the design models for a thorough extra-functional evaluation. Possible refinement of these models can then be performed in order to generate code with preservation of the desired EFPs; this is not achieved by manually editing the generated code. Instead, the prescribed manner to perform modifications is to edit the design models until generated code conforms with both desired behaviour and specification of EFPs. For clarity’s sake, we refer to preservation of EFPs throughout the development process as the ability to generate implementation code by ensuring that values expected at system specification level and modelled at modelling level are actually matching the ones observed or computed at runtime executing the generated code.

This article discusses motivations and challenges in providing an automated round-trip engineering support for MDE of embedded systems with focus on ensuring preservation of EFPs throughout the entire process, that is to say from modelling to code level. In fact, embedded systems’ resources limitation stresses the criticality of EFPs measurement at code level. This would be the case, e.g., of the EFP “worst-case execution time” for a code-block to be executed on a modern hardware platform with accelerator features like caches, branch-predictors, and translation lookaside buffers (TLBs). This time cannot be tightly bounded until machine instructions and memory allocation have been determined. Whereas early simulation activities can help in evaluating a certain set of properties [8] as well as solving specific issues such as bottlenecks and deadlocks, there still exist EFPs that cannot be precisely evaluated at modelling time. Therefore, this work aims at providing support for those properties and leaves apart simulation and analysis activities performable at modelling level. Precise values are obtained only when the system is run in terms of code on a specific platform; in any case, these values depend on the kind of EFP, its observability and the complexity of the target platform.

Moreover, this work proposes a solution to the implementation of the automated round-trip engineering mechanism: generation of source code entails the creation of trace links between models and code, code execution is monitored, and observed values are appropriately back-propagated to modelling level. In this way, a thorough extra-functional evaluation of the design model is made possible, and the provided automation in the preservation process relieves the developers’ effort of manual activities (e.g., additional testing activities on models, involvement of domain experts).

The combination of full code generation together with the provided cross-cutting back-propagation capabilities makes the contribution of this article 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 [9] the round-trip support had been preliminarily validated on a smaller case study considering code generation and traceability links creation as black-box activities. In this work we adapted the round-trip support to the new CHESS modelling language (CHESS-ML) [10] defined within the Composition with Guarantees for High-integrity Embedded Software Components Assembly (CHESS) project. In order to do so, we had to design and implement appropriate transformations for generating 100% of the code as well as the traceability information, since none of these was already available. Then, in order to perform a thorough validation of the approach and evaluate crucial characteristics such as scalability and reusability, we tested it on an industrial-sized case study; moreover, both modelling framework and target platform were different from the preliminary validation.

The rest of the article 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. Additionally, the problem is formalised in terms of goal, intended contribution and core challenges. In Section 3, a subset of the Asynchronous Transfer Mode (ATM) [11] Adaptation Layer 2 (AAL2) case study is presented and used as running case study throughout the article. The proposed approach is initially described in Section 4 while details about challenges and solutions for each step of the process are unwound in Section 5 and Section 6 where actual implementation of the proposed solutions and application to the running case study are presented too. In Section 7 a final discussion is presented while conclusions and planned future enhancements are proposed in Section 8.

2. Background

The integration of model-driven and component-based (MDCB) processes is meant to help in mastering the ever-increasing complexity of embedded software systems design. In particular, this integration discloses opportunities to reduce costs and risks by:

1. enabling effective modelling of EFPs such as safety, reliability, availability and dependability, to mention a few, and
2. providing automation where applicable in the development process [12].

One of the essential benefits gained by adopting model-driven techniques is correct-by-construction, which opposes to the more costly correct-by-correction typical of code-centric approaches. Correctness-by-construction is intended as the ability to demonstrate or argue the software correctness in terms of the approach exploited to generate it. In this respect it is worth noting that the notion of correctness refers to the adherence of the generated code to what specified at model level, once the generation process (i.e., model transformations) has been validated [5]. Nonetheless the correctness of the user solution in the modelling space must be demonstrated for every model, for instance by adopting model verification methods. Possible model-based analysis techniques employed for this purpose are discussed in Section 3.1, even though the verification of the models goes beyond the scope of our contribution.

In the last years, MDCB has been recognised as extremely promising and the large number of works recently published on this subject gives proof of this research trend [13]; tools and frameworks have been developed for supporting this development process [14–16]. Our approach targets an MDCB development process; more specifically, while component-based software engineering (CBSE) contributes with design patterns2 and mechanisms for modelling the system, MDE provides, through model transformations, the mechanisms for enabling generation of code from models and back-propagation of observed values. While the proposed
solution can be considered MDE-dependent, other paradigms could be adopted to replace CBSE for design patterns and mechanisms.

2.1. Related work

Generally, management of EFPs is considered a core development task; in the embedded domain the ability in carrying out this task has still to be improved [17]. Efforts in dealing with EFPs related to composability can be found in the Web Service domain [18,19]; domain-specific languages and UML profiles have been defined to model EFPs [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 of composability with guarantees 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 in terms of abstraction levels and no automation is provided. It is worth noting that having a mechanism to automatically annotate the design model with monitored values is of critical importance in order to assist the developer in understanding at a glance the relationships between expected behaviour and model entities at design level, without having to inspect the related generated code.

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 observed at code level back to design models for better understanding of those EFPs that cannot be accurately predicted at higher levels of abstraction.

The concept of traceability in software development is generally referred to as a way to store relationships between related artefacts [27]; with respect to MDE, since model transformations are the means 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 the rules involved in their creation. Typically, trace links are exploited for synchronisation purposes or to evaluate the impact of source modifications to the existing generated targets [28]. This article extends the concept of traceability to keep track of implementation code portions corresponding to design model elements in order to be able to update their quality attribute values coming from monitoring of code execution.

System monitoring is a widespread activity to survey applications’ behaviour especially when systems’ failures can have serious impacts or even catastrophic consequences [7]. 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, from which observed values are extracted and used to refine design models with sharpen EFPs’ values detected during the execution of the system.

2.2. Problem statement

In this section problem, intended contribution and core issues to be solved are formalised.

**Goal:** aiding in the evaluation of extra-functional preservation by providing means to enrich the design models with EFPs monitored at run-time.

**Contribution:** an automated round-trip support for the preservation of EFPs throughout an MDCB development of embedded systems. Specific contributions are a code generation mechanism enhanced for enabling back-propagation capabilities by maintaining explicit traceability and consistency between source models and generated code as well as back-propagation of monitoring results to modelling level for thorough evaluation of EFPs.

**Core challenges:** in order to provide a solution for the problem the following challenges must be tackled:

- Generation of 100% of code from design models for achieving correctness-by-construction; in fact, having to write part of the code by hand might invalidate the process. Despite solutions for code generation can be found in the state of the art, implementing a code generator for a new modelling language as well as for a selected target platform poses technical challenges due to the complexity of both.
- Creation and maintenance of traceability information; this information is crucial for ensuring a correct back-propagation of EFPs’ observed values from code to modelling level.
- Decoding of monitoring results to obtain observed values to be back-propagated to modelling level.
- Provision of back-propagation facilities to correctly inject the observed values to the appropriate placeholders in the design models.

3. A running case study: the AAL2 subsystem

The back-propagation had been preliminarily validated on the Advanced Cruise Control subsystem on top of the PrIPE platform using the ProCom component model [9] considering the other activities composing the round-trip support (i.e., code generation and traceability links creation) as black-box activities. In order to evaluate significant characteristics of the approach, such as scalability and reusability, the proposal has been validated against an industrial case study on top of the CHESS framework [10]; the target system is a partial version of the AAL2 industrial subsystem which was originally intended to adapt voice for transmission over ATM and is currently used in telecommunications as part of connectivity platform systems.

3.1. The CHESS framework

The cross-domain modelling language (CHESS-ML) and framework [10] defined within the Composition with Guarantees for High-integrity Embedded Software Components Assembly (CHESS) project are used for the specification of the AAL2 subsystem. CHESS allows the specification of a system together with relevant EFPs 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 design patterns and mechanisms. The CHESS component model is conceived in a manner that permits to domain-specific needs to be addressed by adding specialisation features to a domain-neutral core. In this
way 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 the system’s break-down into components is modelled in the Component View, where functional and extra-functional characteristics of the system are defined in two separated subviews as follows:

- Functional: the development style follows component-based design, by which each component is equipped with provided and required interfaces realised via ports and with state-machines and other standard UML diagrams to express functional behaviour. Moreover, the Action Language for Foundation-UML (ALF) [31] is used to enrich the behavioural description; in this way, we reach the necessary expressive power to be able to generate 100% of code directly from the functional models.
- Extra-functional: the functional models are decorated with extra-functional information. In this way the two pieces of information (i.e., functional and extra-functional) are kept separate thus complying with the principle of separation of concerns adopted in CHESS.

Moreover, the set of design views is completed by: Requirement View, used to model the software requirements and associate them to other model entities, Deployment View, which supports the modelling of the target execution platform and software to hardware components allocations and the Analysis View, which is a set of subviews in which the user can model the analysis contexts which will be used as input for the analysis tools. This view is split in two distinct views, each specialised for a given type of analysis: (i) dependability (Dependability Analysis view) and (ii) predictability (RT Analysis view). Most importantly, for each technique have been implemented back-propagation features for enriching the design models with the analysis results thus enabling a multi-perspective extra-functional evaluation of the system.

State-based dependability analysis, Failure Propagation Transformation Calculus (FPTC), Failure Mode Effects and Criticality (FMECA), Failure Mode and Effect Analysis (FMEA) and Fault Tree Analysis (FTA) are the means through which CHESS supports different kinds of evaluation of the dependability attributes of the system [32,33].

In CHESS the user model can be submitted to static analysis in the extra-functional dimensions of interest. For example, schedulability analysis verifies whether the timing requirements set on interfaces can be met [34]. The extraction of information from the user model (i.e., generation of a Platform-Specific Model, PSM, or a Schedulability Analysis Model, SAM) and generation of the input for the analysis tools are automated; the results of the analysis are propagated back to the design model as read-only attributes of the appropriate design entities. Thanks to the full automation, as well as for the code generation and monitoring described in this work, the analysis can be iterated at will until the designer is satisfied with the result.

### 3.2. Modelling the AAL2 subsystem

The AAL2 subsystem used as case study for the validation of the proposed approach is an industrial-sized telecommunication system composed by several hundred thousands of component instances and multiple levels of hierarchical composition of components. In Fig. 1 we propose a simplified version of the AAL2 subsystem which 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 running example of the actual case study we consider only part of the NCC internal structure while considering AAL2RIClient and NCIClient as stubbed. NCC is a connections handler providing connectivity services for the establishment/release of communication paths between pairs of 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. 2), 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 (NodeConnElem instances).
- **NodeConnElem**: that handles management of connections to the network 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 running case study we consider only the first two levels of decomposition (down to the NCC’s internal structure) since additional levels do not affect the functioning of the approach. CHESS allows the definition of EFPs 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 **RsSpecification** stereotype [21], specifically defined in CHESS for the specification of real-time specific properties. In our case the placeholders used for back-propagation of observed values 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 Figs. 1 and 2 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. 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 this sense. The behavioural definition of the system (NodeConnHandler state-machine in Fig. 3) 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 reduced 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 \texttt{Pl\_NCl\_2\_NCC} interface; this 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). Then it sends a request through the \texttt{RI\_NCC\_2\_AAL2RI} interface for each end-point to their respective \texttt{AAL2RClient} to activate the access to the transport layer. Once both end-points have positively responded through their respective \texttt{RI\_AAL2RI\_2\_NCC} interface, NCC confirms the establishment of the connection to \texttt{NCIClient} through the \texttt{RI\_NCC\_2\_NCI} interface. In Fig. 3, the state-machine describing the behaviour of \texttt{NodeConnHandler} component is shown. ALF code specifying the behaviour of the component’s operation \texttt{sendResponse} (matching the homonymous state-machine’s transition) is also shown in the figure. Communication between components in terms of operations (\texttt{node2clientResponseOk()}, \texttt{node2clientResponseFail()}) called on required ports (\texttt{RI\_NetDisp\_2\_NCC}) is depicted in the action code fragment.

4. A round-trip support for the preservation of EFPs

Once modelling tasks have been completed, we generate target code through appropriate model transformations (Fig. 4a). Information regarding tracing of source (e.g., model elements) and target (e.g., code segments) artefacts has to be defined and maintained for further back-propagation activities. Therefore, code generation transformations have to be properly defined by encoding apposite rules for the generation of explicit traceability links [35] between models and code (Fig. 4b). These rules are in charge of populating the back-propagation model with traceability information according to the meta-model depicted in Fig. 5. Once code and traceability links have been generated, EFPs can be evaluated by selected code execution monitoring or code static analysis tools (Fig. 4c). Depending on the capabilities of these tools and their output format, different actions, varying from text-to-model to model-to-model transformations (Fig. 4d), are required to extract and formalise EFPs’ observed values in order to have a complete trace chain from models to values. The last step of the round-trip approach aims at annotating the design models with the observed values (Fig. 4e) through dedicated model-to-model transformations.

As depicted in Fig. 4, the back-propagation model, as it contains information concerning both traceability and observed values, can be considered the core artefact in the process. Moreover, 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 would be used for monitoring and analysis activities. An alternative solution could have been to back-propagate the results of those activities directly to the design models; this solution demands a different back-propagating transformation for each involved monitoring tool. Using an intermediate back-propagation model reduces the approach adaptation overhead caused by the adoption of different monitoring tools. In fact, only ad hoc transformations from monitoring results to the back-propagation model have to be provided; transforming directly from these results to design models would be more intricate due to modelling languages’ complexity.
Fig. 2. NCC composite structure in CHESS.

Fig. 3. NodeConnHandler state-machine in CHESS.
The proposed approach is decomposed in two fundamental issues, namely (i) how to generate and store trace links between design models and generated code, and (ii) how to retrieve useful information from monitoring of code execution and propagate the observed values back to the design models. Challenges and proposed solutions, together with their application to a running example, are discussed in the next sections.

5. Generation of executable code and traceability links

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 forward-backward transformations, (ii) links derived from code analysis, (iii) inferred links computed on the basis of change management of system’s items [27]; in our work we consider explicit links derived from transformations as described in Section 5.1.2.

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 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 appropriate rules for the generation of explicit traceability links between source and target. In this way, information exchanged among models through transformations is formally stored and maintained in structures that are easily and deterministically interpreted and information pieces reachable following precise patterns.

The most appropriate structure in our case is a model, here called back-propagation model, which conforms to the back-propagation meta-model depicted in Fig. 5; a similar structure for storing tracing information can be found in [36]. The back-propagation meta-model has been defined for enabling the creation of back-propagation models which are able 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) observed values. 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 observed values to the design models. A trace element TE (TraceElement in Fig. 5) 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. 5) contained in a design model SM (SourceModel in Fig. 5), EU is an executable unit (ExecutableUnit in Fig. 5) contained in an executable entity EE (ExecutableEntity in Fig. 5), and MEP (ModelElementProperty in Fig. 5) is an extra-functional property defined for ME and calculated by monitoring the execution of EU. A typical case of this 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. 5) defined in the model element specification ME and FUP (FunctionalUnitProperty in Fig. 5) which represents an EFP defined for FU and meant to be calculated by monitoring the execution of EU. A typical case of this granularity is the operation level in the component definition in a component-based architecture.

More generally, the back-propagation model BM (BackpropagationModel in Fig. 5) is a triple $<$TE*, SM, EE*> where TE* is a non-empty set of trace elements, SM is a design 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 on the needs, an executable unit could be defined in more detail by adding information about start and end point within the code file (i.e., executable entity).

Apart from the traceability links, the back-propagation model hosts the information extrapolated from the monitoring activities. More specifically, each property $P$ defined in BM has a property value $V$ (PropertyValue in Fig. 5), which is calculated during monitoring activities and represents the value to be propagated back to the related extra-functional annotation’s placeholder in the design model.

5.1. Applying the solution to the AAL2 subsystem

The code generation process returns two products: (i) generated full target code and (ii) back-propagation model containing trace links between design model and produced code. In this section we demonstrate how the proposed approach has been validated against the AAL2 subsystem in terms of generation of executable C++ code and explicit traceability links.

5.1.1. Full 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++ code from the design
models defined in CHESS-ML. Given the design models, our solution aims at providing a full code generation that entails both structural and behavioural descriptions 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; the details about the code generation go beyond the scope of this article. The code generation process relies on models and transformations as main actors for the generation of target code and is composed by the following tasks:

a. A set of model-to-model transformations, defined using the Operational QVT (QVTo) transformation language, act on the CHESS model to generate two different intermediate artefacts:
   - Instance model: that instantiates components and ports according to their cardinality 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 design 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 this step is given in the next Section (5.1.2).

b. 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).

c. Finally the C++ implementation code is generated through model-to-text transformations, defined using the statically-typed template Xpand language [38], taking as input the sole intermediate model (Fig. 6c).

5.1.2. 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 modelling elements (i.e., components, ports, operations and EFPs) and code execution units in terms of functions 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 [39].

Navigating the CHESS model from the root component through all its composition levels, for each component instance a number of trace elements are created to keep track of the properties monitored at operation level. 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 QVTo transformation in charge of creating the trace elements, related to each of the instantiated component instances operates according to the pseudo-code in Fig. 7.

More specifically, the algorithm is composed by three main steps:

---

6 The QVT language, abbreviation of Query/View/Transformation, is defined [37] 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.
b. for each operation defined in the contained component, an element of type 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 containment 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 identities; 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 this information in order to allow correct injection of the observed values to the right placeholder in the back-propagation model.

c. finally a TraceElement is created for each of the properties to be monitored according to the definition previously given in this section.

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 this trace can be summarised as follows: the trace element client2NodeConnect._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. The fragment of the back-propagation model that we consider for the next back-propagation steps is depicted in Fig. 9.

6. Code execution and back-propagation of observed values

Once the target code has been automatically generated as well as the traceability links, selected EFPs can be measured by monitoring the code execution on a specific platform. At this point, independently from the analysis or monitoring tool used for the measuring activities, the EFPs can be evaluated comparing expected with observed values; for this reason the latter are to be propagated back to the design models. In order to perform back-propagation we need to be able to navigate through the development artefacts from design models down to observed values passing through the generated code. Therefore defining and maintaining explicit traceability links between models and generated code is only the first ring of the needed traceability chain. In fact, traceability between code segments and observed values has to be defined and maintained too. For this purpose, additional actions to manipulate monitoring results for extracting and storing values in proper structures are needed.

For the provision of back-propagation capabilities the approach fights against well-known reverse engineering challenges in mapping data models derived from data analysis to more abstract conceptual design levels. This is usually achieved by supporting iteration of the process and bidirectional mapping from models to analysis data models and vice versa [40]. Our solution achieves back-propagation through a set of model-to-model transformations which enrich the design models with the EFPs’ observed values in proper structures are needed.
values gathered at code level by monitoring activities. The back-propagation process can be decomposed as follows:

- Monitoring results and traceability information management: 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 need to be manipulated in order to extract the observed values and store them in formal structures to be fed to the back-propagating transformations. The proposed solution provides storing structures as part of the back-propagation meta-model. Observed values as source for the back-propagating transformations are not enough. In fact, the traceability chain defined along the path from design models to observed values is also part of the source artefacts to be fed to the transformations in order to correctly propagate values back to the design models. Moreover, regarding the code inserts needed for monitoring activities, they are automatically generated with the rest of the implementation, thus not jeopardising the consistency between source models and code. Additionally, the computation, both space and time wise, of the EFPs considered in this paper (i.e., execution time and allocated memory) did not affect the overall performances of the system. Nevertheless, the monitoring and computation of more complex EFPs as well as the employment of a more complex target platform may be needing complex calculations through specific techniques and tools as well as the use of additional modelling artefacts; this might influence the overall performances of the system. Specific effort will be directed to these more complex cases (please refer to Section 8).

- Annotation of design models: the very final step of our approach is the actual enrichment of the design models with EFPs' values gathered during code execution monitoring activities. The enrichment should be performed by injecting the observed values into the related model elements' placeholders at modelling level. The complexity of the model transformation process in charge of the injection depends on the modelling language’s capabilities in modelling EFPs.

Once completed, the back-propagation task produces an extra-functionally enriched version of the design models. At this point it is possible for the developers to evaluate these models and possibly edit them when needed. The process might require multiple iterations in order to reach the desired quality level, in terms of EFPs, required by the system specification.

6.1. Applying the solution to the AAL2 subsystem

In this section our running case study is used to show the conclusive phases of the approach in terms of: (i) monitoring activities and management of monitoring results as well as traceability information, and (ii) propagation of observed values back to the design model in CHESS.

6.1.1. Monitoring results and traceability information management

In this work we implement the injection by means of a text-to-model transformation since the monitoring activities give a textual
More specifically, they produce a log file as a set of four-token lines formatted as follows:

*ExecutableUnit ModelElementInstance.id Property Value*

Once C++ code is generated from our AAL2 subsystem design models, we use the Linux API `getrusage` for monitoring its execution. The resulting monitoring log file is depicted in Fig. 10.

The injecting text-to-model transformation is implemented in Java and, taking as input the back-propagation model BM and the monitoring log file MF, acts according to the algorithm in Fig. 11 described in the followings.

**Fig. 9.** Portion of the generated back-propagation model related to the AAL2 subsystem.

**Fig. 10.** Monitoring log file.
for each line l in MF do
    execUnit = l[1];
    id = l[2];
    property = l[3];
    value = l[4];
    BMtrace = BM.search(execUnit, id, property);
    if BMtrace! = NULL then
        BMtrace.property.value = value;
    end if
end for

Fig. 11. Injection algorithm.

code the observed value is derived and thereby to which trace element has to be injected in the back-propagation model. Once a match is found, which is to say that there is a trace element BMtrace linking Property with ExecutableUnit and Model-ElementInstance in the back-propagation model, then the token Value is injected into the correct placeholder pointed by Property. The resulting complete back-propagation model BM for the AAL2 subsystem is 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.

6.1.2. Performing the back-propagation
Since the information to be back-propagated to the design 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. Taking as input the design model and the back-propagation model, the transformation performs a set of in-place transformations on the design model to enrich it with the observed values stored in the back-propagation model (Fig. 14).

As defined in Section 5, a back-propagation model is composed by a non-empty set of trace elements TE defined as quadruples \(<\text{ME}, \text{FUP}>\) where \(\text{ME}\) is a model element contained in a design model SM. \(\text{FU}\) represents an operation/method defined in \(\text{ME}\) and \(\text{FUP}\) represents an EFP defined for \(\text{FU}\) and observed by

![Fig. 12. Injection of the monitoring results to the back-propagation model.](image-url)
monitoring the execution of the executable unit EU. The transformation algorithm (Fig. 13) takes as input the back-propagation model BM and AAL2 subsystem design model SM defined in CHESS, as well as the meta-models to which they conform to; the output will be an enriched version of SM.

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. In Fig. 15 and Fig. 16 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 design model and it is now possible for the developing team to evaluate the EFPs preservation. Eventually, optimisation activities are performed directly at modelling level rather than at code level with a consequent conservation in terms of consistency among the artefacts and their properties.

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 design 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),

![Fig. 14. Back-propagation transformation.](image)

![Fig. 15. Back-propagation to AAL2 subsystem in CHESS.](image)
is an atomic process kept transparent to the developer who is not able to modify any of the intermediate artefacts.

- Back-propagation: the user has control only over design models and generated code. Modifying any of these artefacts, the developer may cause inconsistencies during the back-propagation phase and hence jeopardize the reliability of the back-propagated values. 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 design models, generated code is not meant to be manually edited.

Validating the approach against an industrial-sized case study and on top of a different framework gave us the possibility to make comparisons with the preliminary validation described in [9] 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 step-wise. Moreover, within the same case study we tested several design 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 (i.e., within 15 min on machine running on a 2.6 GHz CPU and 8 GB RAM) while degrading going toward $10^5$ (i.e., over 30 min on the same machine); 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 obvious and stems from the computational 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 design model) resulted to be less dependent on the design model’s size. Intermediate artefacts’ size may grow proportionally to design 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 these adaptations can vary depending of the chosen modelling language, target implementation code, and so forth; the complexity of the model transformations involved in the process is in fact affected by expressiveness and precision of the involved languages. 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. Addi-
tionally, extending the approach to entail more complex EFPs that need to be calculated with specific techniques and tools as well as using additional modelling artefacts would affect the adaptation overhead too; specific effort will be directed to these more complex cases. It is worth noting that not all EFPs can be statically preserved, hence run-time enforcement may be needed together with code generation; for this purpose we employed the platform adaptations for enforcement of EFPs provided by CHESS.

Moreover, driven by the possibilities given by the CHESS framework in defining EFPs, 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, as described in Section 5. This enhancement makes traceability and back-propagation capabilities more powerful and easily adaptable to a wider set of different modelling frameworks and needs.

Concerning the achievement of correctness-by-construction, the aim of this work is to enhance the CHESS approach, already equipped with model verification techniques in terms of model-based analysis (Section 3.1), by enabling evaluation of certain EFPs at code level and their back-propagation to source models.

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 EFPs preservation from models to generated code. This capability, achieved through generation of 100% of the code and corresponding traceability together with back-propagation from code to modelling level makes the contribution of this article unique.

The approach is meant to aid an MDB development process to preserve functional and 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 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 validated on a selected case study.

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 EFPs (e.g., execution time and allocated memory) measured; appropriate transformations are defined to perform injection of observed values into the back-propagation model. This model is then taken as input by an appropriate model-to-model transformation for propagating the observed values back into the related model elements’ placeholders at modelling level.

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. Moreover, new generation EFPs, such as energy-related properties, should be taken into consideration for future evaluations and possible extensions of the provided round-trip support.

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. 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. A very interesting direction would certainly be to focus on the adaptation of the round-trip support to be applied to the numerous embedded systems entailing heterogeneous platforms.

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 No. 216682) [10].

References