Toward a Round-Trip Support for Model-Driven Engineering of Embedded Systems

Federico Ciccozzi, Antonio Cicchetti, Mikael Sjödin
School of Innovation, Design and Engineering (IDT)
MRTC, Mälardalen University
Västerås, Sweden
email: [federico.ciccozzi, antonio.cicchetti, mikael.sjodin]@mdh.se

Abstract—In a model-driven environment aiming at generating implementation code ensuring that extra-functional properties modeled at design level are preserved at execution time, a full round-trip engineering approach is often needed. Target code is meant to be generated from design models through appropriate model transformations; once the code has been generated, quality attributes of the embedded system are evaluated by execution monitoring/analysis tools. Eventually, in order to complete a model-driven round-trip approach, provision of back-annotation of the target code analysis results to modeling level is crucial for ensuring preservation of analyzed extra-functional aspects. In this work the problem of providing such approach in terms of process and related challenges is described together with a proposed solution. Particular emphasis is put on the description of how both traceability information and code analysis results are formalized in order to enable the desired back-annotating capabilities.

I. INTRODUCTION

Model-Driven Engineering (MDE) aims at facilitating the system development by creating, maintaining and manipulating models. In fact, models provide abstractions of a real phenomena which 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 [1]. Rules and constraints for building the 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 proper 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]. Generation of code to be executed on specific target platforms is too often seen as the final and non-coming back step of an MDE approach [4]. Therefore, quality in terms of extra-functional aspects of the system modeled 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. Then, in order to be able to perform an evaluation of expected against computed extra-functional properties, the abstraction gap (model-code) between them must be settled; this is performed by back-annotating the extra-functional computed values 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.

In this paper we describe motivations, challenges and our solution towards the implementation of 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 modeling to code execution level. In fact, embedded systems’ resources limitation stresses the criticality of extra-functional properties measurement at code execution level. In order to be able to evaluate the measured values against the expected behavior modeled at design level, crucial activity in such approach is the back annotation of code execution monitoring results at modeling level for a complete extra-functional evaluation. Moreover, 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.

The rest of the paper is structured as follows. Section II 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 III the actual proposed approach is described and fully unwound in its details together with challenges and solutions for each step of the process. The paper culminates with sections IV and V in which the actual implementation of the solution and its validation against a selected case-study are reported. Conclusions and planned future enhancements are presented in section VI.

II. BACKGROUND

A. Motivation

Design of embedded software systems requires support capable of managing their ever-increasing complexity. In this direction an integrated model-driven and component-based (MDCB) process would help in handling complexity of such systems and thus reducing costs and risks by: (i) enabling effective modeling 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 [6].

In the last years, the aforementioned integration has been recognized as extremely promising and the large number of works recently published on this subject gives proof of this research trend [7]; tools and frameworks have been developed for supporting such development process [8], [9], [10]. The focus of our approach within a MDCB process is to provide a round-trip support for aiding in ensuring a required level of quality preservation in terms of extra-functional properties from the modeling artifacts to the generated code. Proper description, verification and preservation of such properties throughout the development process would 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 [11], [12].

While early simulation activities can help in evaluating a certain set of properties [13] as well as solving specific issues such as bottlenecks and deadlocks, other extra-functional properties can not be evaluated at modeling time; in this work we focus on such properties, and therefore leave apart simulation and analysis activities performable at modeling level. In fact, especially in the embedded domain, resources limitation sharpens the criticality of having extra-functional properties evaluation at code execution 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 behaviors of their variations. Therefore, it is not possible to have precise values until the design is run in terms of code on a specific platform. As a consequence, some of the information and values at design level, used also for validating the design itself, can only be gathered at code execution time. Hence, a round-trip support for enabling back-annotation of such values to the source model aids the MDBC process in achieving automated generation of code with preservation of extra-functional properties across all the abstraction layers.

B. Related Works

Generally, management of extra-functional properties is considered a core development task; in the embedded domain such activity has still to be improved [14]. Efforts in dealing with extra-functional aspects of composition can be found in the Web Service domain [15], [16]; domain-specific languages and UML profiles have been defined to model extra-functional aspects [17], [18]. 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 [19]; nevertheless, effective solutions for composability with guarantees in embedded real-time domain are still missing. Other attempts propose to solve the problem by back annotation, i.e. by reporting the values measured at the execution time back to the models in order to possibly fix and/or refine estimated numbers. Navabi et al. in [20] in the early 90’s, and some years later Mahadevan and Armstrong in [21], carried out different approaches for back-annotating behavioral descriptions with timing information; however, both of those works operate horizontally in terms of abstraction level 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 [22] back-annotation for enabling execution traces retrieved by model checkers or simulation tools to be integrated and replayed in modeling 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 the one we propose is described by Guerra et al. in [23] where back-annotation of analysis results to the original model by means of triple graphical patterns is described. Nevertheless, the approach is meant to operate at platform-independent modeling level with back-annotation between models. While, dealing with embedded real-time systems, our approach focuses on back-annotating analysis results computed at code execution level to design models at platform-aware modeling level for better understanding of those extra-functional properties that can not be evaluated at higher abstraction levels.

C. Contribution

This work illustrates a general technique whose goal is to provide support in terms of tracing/back-annotating features for an MDCB process to achieve preservation of those extra-functional properties which can not be predicted statically at modeling 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 such a platform, registered values are automatically annotated back to the modeling 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 annotation activities; in turn, design models can be evaluated and possibly refined to achieve extra-functional properties preservation and hence correctness-by-construction [5]; 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 never 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 aspects made possible by the back-annotation.

In the following, the proposed approach is decomposed in three fundamental issues, namely how to store trace links between source models and generated code, how to retrieve
useful information from monitored code execution results, and how to annotate the registered values back to the source models; for each of them, challenges and possible solutions, together with their implementation, are discussed.

III. PROPOSED APPROACH

In this work we propose a round-trip support for MDCB processes which aim at solving the challenge of generating implementation code ensuring that extra-functional concerns modeled at design level are preserved at code execution level (Fig. 1). Once design modeling tasks have been successfully completed, the objective is to enable automatic generation of target platform-tailored code from source models. Taking design models as source artifacts, 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)) has to be defined and maintained for further back-annotation activities. Therefore, code generating transformations have to be properly defined by encoding apposite rules for the generation of traceability links (explicit traceability [24]) between models and code (Fig. 1b); such rules populate the back-annotation 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, 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 complete information chain from models to monitoring results.

The last step of the round-trip approach aims at eventually annotating the source models with the code execution results (Fig. 1e) through model-to-model transformations.

As depicted in Fig. 1, by containing information concerning both traceability and monitoring results, the back-annotation model can be considered the core artifact in the process. That is to say that, as long as the modeling language does not change, the transformations performing the back-annotation will not need to be modified, even if different tools are used for monitoring activities. An alternative solution could indeed be to back-annotate the monitoring results directly to the model with the support of a traceability model; in this case for each monitoring tool a different back-annotating transformation should be provided. Using an intermediate back-annotating 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-annotation model, which are generally much less intricate than transformations from monitoring results to source models due to modeling languages’ complexity, have to provided.

A. Code Generation and Traceability

In the proposed approach, the task of automating the generation of platform-tailored 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 further back-annotation activities. Traceability can be any relationship existing between artifacts 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 [25]. Our approach relies on models and transformations as main artifacts; 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. Tracing information is then maintained in appropriate structures, such as a back-annotation meta-model (Fig. 1) providing concepts for maintaining traceability between source models and generated code; a similar structure for storing tracing information can be found in [26].

B. 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 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 by comparing their estimated and measured values;
for this reason measured values are to be back-annotated to design models. In order to perform such back-annotation we need to be able to walk through the development artifacts 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 properly structure such information for easing back-annotation activities are needed.

C. Back-annotation to Source Models

Back-annotating monitoring results to modeling level represents the last step of the approach, crucial for evaluating and consequently optimizing the design models for ensuring preservation of analyzed extra-functional aspects. 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 level by supporting iteration of the process and bidirectional mapping from models to analysis data models and vice versa [27]. Back-annotation is performed through appropriate model-to-model transformations which annotate the targeted source models with the required information about quality attributes derived from the monitoring results.

This activity can be decomposed as follows:

- Monitoring results representation: results coming from the monitored execution of the generated code are part of the source artifacts for back-annotating 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-annotating transformations. The proposed solution provides storing structures as part of the back-annotation meta-model.

- Tracing information management: giving monitoring results as source for the back-annotating 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 source artifacts to be fed to the transformations in order to correctly back-annotate results 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-annotation meta-model.

- Annotation of design models: the very final step of our approach is the actual enrichment of source models with information concerning evaluation of extra-functional properties derived from the code execution monitoring activities. The enrichment should be performed by injecting the computed property values into the related modeled elements at modeling level for completing the design models with analysis information. The effort to be put in such injection depends on the modeling language’s capabilities in modeling extra-functional properties.

Even though traceability has been correctly defined and maintained, there may still be the possibility of missing correspondences between monitored values and those parts of the modeled system that cause such values to come about; in this respect, possible solutions for minimizing such missing correspondences will be evaluated later on in the approach development. Once completed, the back-annotation activity produces an extra-functionally evaluated version of the system models. At this point it is possible for the developers to validate the system and eventually perform further optimization activities on the models. The process might require several iterations in order to reach the desired level of service, in terms of extra-functional properties, required by system specification.

IV. IMPLEMENTING THE ROUND-TRIP SUPPORT

A. Back-annotation Meta-Model

The round-trip support we propose is based on models and transformations as main artifacts. Therefore information exchanged among models through transformations must be 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-annotation model, which conforms to the back-annotation meta-model depicted in Fig. 2. The back-annotation meta-model has been defined for enabling the creation of back-annotation models which could store the information gathered during code generation and monitoring tasks in an ordered and formal manner.

Conceptually, two classes of information are stored in the back-annotation 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-annotation meta-model is indeed the trace element, which allows the navigation through all the stored information needed for back-annotating the source model with monitoring results. A trace element $TE$ (TraceElement in Fig. 2) is a triple $<ME, EU, P>$ 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 $P$ (Property in Fig. 2) is an extra-functional property modeled in $ME$ and monitored by $EU$. More generally, the back-annotation model (conforming to the back-annotation meta-model) $BM$ (BackannotationModel in Fig. 2) is a triple $<TE^*, SM, EE^*>$, where $TE^*$ is a nonempty set of trace elements, $SM$ is a source model (i.e. a composition of model elements), and $EE^*$ is a nonempty set of executable entities which are in turn composed by executable units. Depending from the code generation and the monitoring activities, an executable unit could even be more specifically defined as a code block with...
Apart from the traceability links, the back-annotation model has to be able to host the information extrapolated from the monitoring activities and that would complete the information ring model-code-results needed for back-annotating the monitored properties computed values into the source model. In the back-annotation 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 back-annotated to the related property in the source model.

### B. Code Generation and Traceability Management

As aforementioned, in order to be able to perform back-annotation operations, a full trace from source model to relative generated execution entity/ies (i.e. code) and monitoring results has 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 transformation responsible for the code generation. Such rules create a mapping between each model element to its respective generated execution unit or code block by creating and populating a back-annotation model according to the constraints defined in the back-annotation meta-model. Hence, the code generation process gives two artifacts as output: (i) generated target code and (ii) back-annotation model containing trace links between source model and produced code.

In this work we do not focus on how the code is generated and the traceability information are stored in the back-annotation model, but rather consider it as black-box activity and focus on the feasibility and usefulness of our round-trip in terms of the core back-annotation activities. Future enhancements discussed in section VI will include a validation on an industrial-sized case-study in the context of the CHESS project (cf. Acknowledgments), where code generation is meant to facilitate crucial back-annotation activities [28].

### C. 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-annotation model, the information about property values calculated during monitoring activities has to be injected in the back-annotation model in order to have the complete chain model-code-values needed for back-annotating the source model. The way to perform such injection strictly 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-annotation model. This can be considered the variable point of the whole round-trip support approach in the sense that it can not be generalized for a multitude of different tools, but rather adapted to the output of each of them. In this work we implement it 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 three-token lines formatted as follows: 

\[ \text{ExecutableUnit Property Value} \]

The transformation itself is implemented using Java and, taking as input the back-annotation model $BM$ and a monitoring output file $MF$, acts as follows:

```java
for each line $l$ in $MF$ do
    for each token $t$ in $l$ do
        if $n == 1$ then
            ...
```

Fig. 2. Back-Annotation Meta-Model
\text{execUnit} = t;
\text{else if } n == 2 \text{ then }
\text{property} = t;
\text{end if}
\text{else if } n == 3 \text{ then }
\text{value} = t;
\text{end if}
\text{end if}
\text{end for}
\text{BMtrace} = \text{BM.search(execUnit, property)};
\text{if } \text{BMtrace}! = \text{NULL} \text{ then }
\text{BMtrace.property.value} = \text{value};
\text{end if}
\text{end for}

\text{MF} \text{ is navigated line by line, and each of which is tokenized according to the defined format \text{ExecutableUnit Property Value}; the tokens \text{ExecutableUnit} and \text{Property} represent the information pair for which a match has to be sought in \text{BM}. Once a match is found, the token Value is inserted into the \text{BM} to add the calculated value to the related property.}

\section{D. Back-annotating the Source Model}

Once the back-annotation model has been completed with property values derived from the monitoring activities, the final step of the process, that is to say back-annotating monitoring results to the source model by injection can be performed. This task is crucial for enabling models optimization. Since the information to be back-annotated to the source model is in turn stored in a model (i.e. the back-annotation model), the injection is performed by a model-to-model transformation that, taking as input the source model and the back-annotation model, operates a set of in-place transformations [3] on the source model by enriching it with the property values stored in the back-annotation model (Fig. 3). The transformation language used for implementing the back-annotation is Operational QVT (QVTo), more specifically using the Eclipse implementation [29]. The QVT language, abbreviation of Query/View/Transformation, is defined [30] 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.

As aforementioned a back-annotation model is composed by a non-empty set of trace elements and that a trace element \text{TE} is a triple <\text{ME}, \text{EU}, \text{P}> where \text{ME} is a model element contained in a source model \text{SM}, \text{EU} is an executable unit contained in an executable entity \text{EE}, and \text{P} is an extra-functional property modeled in \text{ME} and monitored by \text{EU}. The transformation (Fig. 4) takes as input the back-annotation model \text{BM} and source model \text{SM}, plus the meta-models to which they conform to, and gives as output an enriched version of \text{SM} achieved by means of in-place transformation (i.e. \text{SM} is both input and output of the transformation) acting as described in the following pseudocode:

\begin{verbatim}
for each trace element \text{TE} in \text{BM} do
for each property tuple \text{(ME, P)} in \text{TE} do
\text{SMproperty} = \text{SM.search(ME, P)};
if \text{SMproperty}! = \text{NULL} then
\text{SMproperty.value} = \text{P.value};
end if
end for
end for
\end{verbatim}
specific component model called ProCom [31], especially designed to support the specific development concerns of distributed embedded real-time systems. One of the main objectives behind ProCom is to facilitate analysis of the extra-functional properties typically found in embedded systems such as resource usage, timing and dependability, particularly for enabling early analysis capabilities. In ProCom, components are rich design units that aggregate functional interfaces, modeling, analysis and implementation artifacts. The ACC system is modeled as a ProCom composite component and therefore follows a pipes-and-filters architectural style that separates data and control flows. Data input and output ports are denoted by small rectangles whereas trigger ports are denoted by triangles. Moreover, since the properties to be monitored and back-annotated will be system execution time and the maximum resident memory set size, each component has two attributes (i) `execTime` and (ii) `memory` defined for hosting back-annotated values (Fig. 7).

For more information about ProCom, refer to [32]. The PrIDE tool [33] supports development of systems using ProCom component models and it has been used for developing this system.

### A. From the ACC System Model to Generated Code and Back-annotation Model

Once the system has been modeled in terms of ProCom components and the corresponding C code is automatically generated through ad-hoc capabilities provided by the PrIDE tool, the round-trip support can take over for monitoring code execution and back-annotate the deriving results to the ACC System model. As aforementioned, in this work we do not focus on how the code is generated and the traceability information is stored in the back-annotation model.

The result of code generation activities is a set of C code files per component and the portion of the back-annotation model containing traceability information between modeling elements (i.e. components and attributes) and code execution units in terms of the code artifact implementing their main functionality. Recalling the definition of back-annotation model given in section IV-A, the generated back-annotation model will be the following:

\[
\text{ACC\_back\_Model} = \{\text{T1=T1}, \text{T2=T2}, \text{T3=T3}, \text{T4=T4}\}
\]

where:

\[
\begin{align*}
\text{T1} &= \langle \text{ACC Controller}, \text{accControlImpl, memory}\rangle, \\
\text{T2} &= \langle \text{ACC Controller}, \text{accControlImpl, execTime}\rangle, \\
\text{T3} &= \langle \text{Speed Limit, speedLimitImpl, execTime}\rangle, \\
\text{T4} &= \langle \text{Speed Limit, speedLimitImpl, memory}\rangle
\end{align*}
\]

are the stored trace links (in this example the `ACC Controller` and `Speed Limit` components are considered for the monitoring activities). Since the monitoring activities have not been performed yet, the properties stored in the back-annotation model (Fig. 6) do not have an actual value yet.

### B. Code Execution Monitoring and Back-annotation to the ACC System Model

The code produced is C and the Linux API `getrusage` [34] 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 IV-C. In our example information regarding monitored properties for the components `ACC Controller` and `Speed Limit` are stored in the monitoring result file, after code execution, as follows:

\[
\begin{align*}
\text{speedLimitimpl memory} &= 2040 \\
\text{speedLimitimpl execTime} &= 28860 \\
\text{accControllerImpl execTime} &= 83200 \\
\text{accControllerImpl memory} &= 6600
\end{align*}
\]

Once the execution is completed, the results in terms of monitored properties values are injected into the back-annotation
model through the Java transformation described in section IV-C as shown in Fig. 6. At this point all the information needed for back-annotating the ACC system model is stored in the back-annotation model. In order to eventually complete the round-trip path, these information are to be injected in the apposite ACC system model elements through the QVTo back-annotation transformation, described in section IV-D, fed with the ACC system ProCom model and the back-annotation model as input artifacts. The transformation operates a set of in-place transformations of the source ACC system model giving as output an enriched version of the same model; the properties monitored during the code execution are now equipped with the related resulting monitored value. The ACC Controller component with back-annotated values for execution time and memory is depicted in Fig. 7. The round-trip process has produced an extra-functionally evaluated version of the system models and it is now possible for the developing team to validate the system and eventually perform further optimization activities directly at modeling level rather than at code level with a consequent conservation in terms of consistency among the artifacts and their properties at different abstraction levels.

VI. CONCLUSION

In this work we lay foundations and motivations for a round-trip engineering approach which aims at supporting an MDCB development process cycle in ensuring extra-functional properties preservation. 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 proven by applying them to a selected case-study.

The proposed approach will be enhanced by taking into account the computation of more complex extra-functional properties and providing full automation that would make the intermediate steps of the process to be transparent to the final user. Moreover, management and evaluation of properties like safety and security, which usually differ from properties measurable by means computed values, is planned to be supported by the approach. The enhanced version of the round-trip support will be enriched with its own code and trace links generators in order to be validated at a larger scale within the CHESS project by taking an industrial sized case-study from the telecommunication domain, and the CHESS-ML as modeling language as soon it will be finalized.

The design models will be modeled using the CHESS-ML where the system is defined in terms of components and component model which explicitly declare each required extra-functional aspect in terms of annotations to their functional interface. From these modeled artifacts, C++ code (and trace links in terms of a back-annotation model) will be automatically generated and its execution monitored and analyzed through an appropriate set of tools; apposite transformations will be provided to inject the results into the back-annotation model. Such model will be then taken as input by an evolved version of the model-to-model transformation shown in Section IV-D for back annotating the computed property values into the related modeled elements at modeling level for completing the design models with analysis information. At the end of this process, the developer will get a set of enriched source models from which s/he will be able to extract precious information about the modeled system in terms of monitored
quality attributes and eventually optimize the models to reach code generation by preserving extra-functional properties.

Examining the reusability degree of the main artifacts composing the proposed round-trip approach we can conclude that: (i) the back-annotation meta-model is fully reusable due to its independence from specific languages or meta-models, while (ii) the proposed back-annotation QVTo transformation will have to be slightly adapted, as well as the creation of the traceability links during the code generation, to be able to take as input and navigate source design models conforming to a meta-model (e.g. CHESS-ML) different from ProCom.

ACKNOWLEDGMENTS

The research work presented in this paper has been supported by the CHESS project (ARTEMIS JU grant nr. 216682 - http://chess-project.ning.com/). The authors would like to thank Thomas Leveque for his support.

REFERENCES

[34] Linux Die. Linux 2.6.9 Manual, 2006.