ABSTRACT
Synergies between model-driven and component-based software engineering have been indicated as promising to mitigate complexity in development of embedded systems. In this work we evaluate the usefulness of a model-driven round-trip approach to aid deployment optimization in the development of embedded component-based systems. The round-trip approach is composed of the following steps: modelling the system, generation of full code from the models, execution and monitoring the code execution, and finally back-propagation of monitored values to the models.

We illustrate the usefulness of the round-trip approach exploiting an industrial case-study from the telecom-domain. We use a code-generator that can realise different deployment strategies, as well as special monitoring code injected into the generated code, and monitoring primitives defined at operating system level. Given this infrastructure we can evaluate extra-functional properties of the system and thus compare different deployment strategies.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics—performance measures; D.2.11 [Software Engineering]: Software Architectures—languages

General Terms
Design, Measurement, Performance, Theory

Keywords
component-based software engineering, model-driven engineering, embedded systems, back propagation, deployment optimization, code generation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1. INTRODUCTION
Complexity of embedded systems is continuously increasing and therefore solicits the introduction of more powerful and automated development mechanisms able to mitigate it. In this direction, Model-Driven Engineering (MDE) [1] and Component-Based Software Engineering (CBSE) [2] can be considered as two orthogonal ways of reducing development complexity through different means. The former shifts the focus of the development from hand-written code to models from which the implementation is meant to be automatically generated through the exploitation of model transformations. The latter breaks down the set of desired features and their intricacy into smaller replaceable sub-modules, namely components, starting from which the application can be built-up and incrementally enhanced. Moreover, their combination has been recognised as an enabler for them to definitely break through for industrial development of embedded systems [3].

In this research work we exploit the synergy between MDE and CBSE to demonstrate the benefits of an automated round-trip support in aiding deployment optimization when developing embedded systems. The round trip support consists of the following steps:

- Modelling: the first step is represented by modelling the system through a structural design, in terms of components, a behavioural description by means of state-machines and action code, as well as a deployment model describing the allocation of software components to operating system’s processes;
- Code generation: from the information contained in the design model, we automatically generate full functional code. Note that in the paper we refer to generated code as full or full-fledged if it is entirely generated in an automated manner and does not require manual tuning in order to be executed on the selected platform;
- Monitoring: after the code has been generated we monitor its execution on top of the target platform and measure selected Extra-Functional Properties (EFPs). In fact, in modern complex embedded systems, certain EFPs cannot be accurately predicted at modelling level, hence requiring measurements at runtime. This would be the case, e.g., of performance-related EFPs, that often only emerge in a running product. As example, let us consider two sorting algorithms that speed up a program because they use a big portion of the main memory. Although both increase the performance in isolation and they have no direct func-
tional interaction, in combination they may degrade the overall performance because both share the same (too small) main memory [4]:

+ Back-propagation: at this point, gathered values are back-propagated to the design model and, after their evaluation, the deployment configuration can be manually tuned to generate more resource-efficient code.

The round-trip support and its usefulness in aiding the preservation of EFPs\(^1\) from models to generated code have been already proven and discussed in [5] and [6]. Nevertheless, their limited support for platform configurations (i.e., only single-process) prevented their employment for deployment issues. The approach has been therefore enhanced in order to enable synthesis of design models to either a single-process or to a set of communicating processes. This gives flexibility to generate either highly resource efficient (in terms of inter- system communications) single-process systems or exploit multi-process configurations that could, e.g., run in parallel on multicore and/or maintain error encapsulation within one process.

In this work we describe the enhancements made to the round-trip support and how it can employ measurements gathered at system implementation (or runtime) level towards deployment optimization when developing embedded systems. More specifically, in order to enhance the generation of full code from models addressing multiprocess applications, additional modelling artefacts (e.g., deployment model) had to be considered, higher variability of the transformation process had to be addressed by enhancing the code generation transformations as well as expanding the intermediate metamodels to entail deployment information and diverse communication patterns. In order to employ the approach for deployment assessment the execution platform had to be modified for enabling monitoring facilities and back-propagating transformations were implemented too. Moreover, the approach has been validated against industrial case-studies from the telecom-domain.

The remainder of the paper is structured as follows. The scope of the paper is defined in terms of context delimitation and contribution formalisation in Section 2. The state-of-the-art related to similar approaches with focus on backpropagation features, monitoring activities and deployment optimization based on measurements at system implementation level is described in Section 3. A running example in terms of an industrial case-study is introduced in Section 4 while the proposed solution is described in Section 5. The application of the proposed approach to the example is described in all its details in Section 6 with focus on showing how the round-trip support can aid the developer in taking deployment decisions. Section 7 proposes a discussion on the proposed solution and possible enhancements; the paper is then concluded by a summary in Section 8.

2. CONTEXT

Following the MDE paradigm, a system is developed by designing models and refining them starting from higher and moving to lower levels of abstraction until code is generated; refinements are performed through transformations between models. A model transformation translates a source model to a target model while preserving their well-formedness [7]. Since a model is an abstraction of the system under development, rules and constraints for building it have to be properly described through a corresponding language definition. In this respect, a metamodel describes the set of available concepts and well-formedness rules a correct model must conform to [8].

Since different nuances of the CBSE-related terminology can be found in the literature, in this work we refer to component-based development as prescribed by the UML Superstructure [9]. That is to say, a system is modelled as an assembly of components communicating via required and provided interfaces exposed by ports, where a port represents an interaction between a classifier instance and its internal or external environment. Additionally, features owned by required interfaces are meant to be offered by one or more instances of the owning classifier to one or more instances of the classifiers in its internal or external environment.

A fairly wide variation of different approaches to the measurement of EFPs at system implementation level exists. In this work we focus on runtime monitoring, that represents a method to observe the execution of a system in order to determine whether its actual behaviour is in compliance with the intended one. In comparison to other verification techniques such as static analysis, model checking, testing and theorem proving which are used mainly to determine “universal correctness” of software systems, runtime monitoring focuses on each instance and current execution of a system [10].

In the following sections we describe the scope of the proposed solution in terms of modelling language, target platform and intended contribution.

2.1 CHESS Modelling Language

In our work we employ the CHESS modelling language (CHESS-ML) [11], defined within the CHESS project (cf. Acknowledgements) as a UML profile, including tailored subsets of SysML and MARTE profiles. CHESS-ML 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 development as prescribed by the UML Superstructure [9].

According to the CHESS methodology, functional and extra-functional characteristics of the system are defined in specific separated views as follows:

+ Functional: UML component and composite component diagrams are employed to model the structural aspects of the system while state-machines and activity diagrams are used to express functional behaviour. Action Language for Foundational UML (ALF) [12] is used to enrich the behavioural description. In this way, we reach the necessary expressive power to be able to generate full implementation code directly from the functional models with no need for manual fine-tuning of the code after its generation;

+ Extra-functional: in compliance with the principle of separation of concerns adopted in CHESS, the functional models are decorated with extra-functional information thereby ensuring that the definition of the functional entities is not altered. In respect to the

\(^1\) By preservation of EFPs we intend keeping them within their validity ranges and protect them from violation at runtime.
back-propagation proposed in this work, it is worth clarifying that there is no cloning nor versioning of the design model but rather a decoration in terms of values updates on the extra-functional stereotypes. This means that, at each iteration, the EFPs are updated and, if any modification to the model is manually performed, no history of previous versions is kept.

2.2 OSE Real-Time Operating System

OSE is a commercial and industrial real-time operating system developed by Enea [13] which has been designed from the ground specifically for fault-tolerant and distributed systems. It is widely adopted mainly in telecom-domain and systems ranging from mobile phones to radio base stations [14]. OSE provides the concept of direct and asynchronous message passing for communication and synchronisation between tasks, and its programming model is based on this concept. This allows tasks to run on different processors or cores, utilising the same message-based communication model as on a single processor. This programming model provides the advantage of avoiding the use of shared memory among tasks. In OSE, the runnable real-time entity equivalent to a task is called process, and the messages that are passed between processes are referred to as signals (thus, the terms process and task in this paper can be considered synonyms).

A system modelled through the CHESS-ML is manipulated through model transformations to generate C++ code which constitutes definition and functionality of the processes that will run on the platform. To retrieve information about desired EFPs, the platform has been extended with a set of monitoring and logging processes which collect information about the behaviour of the generated code in terms of memory and CPU usage of a process, execution time, response time, number of signals generated by a process (to mention a few), as well as several system-level properties such as overall CPU usage and total number of signals in the system.

2.3 Contribution

Goal.

The aim of this work is to lessen the developer’s effort devoted to optimise deployment configuration in the development of component-based embedded systems exploiting measurements gathered at system implementation level and avoiding manual editing of code.

Solution.

The solution we propose is represented by a fully automated model-driven round-trip approach that enables:

- Generation of full-fledged code from source models entailing customisable single- and multi-process deployment configurations;
- Monitoring of code execution for measuring selected EFPs at system implementation level, such as CPU load, number of generated signals, system throughput, execution time, response time of a process, instance execution time of a process, heap and stack usage;
- Back-propagation facilities to enrich the source models with values gathered by monitoring.

Through the proposed approach, the developer exploits model-driven techniques, thus operating exclusively at modelling level, and at the same time takes advantages of measurements gathered at runtime and automatically brought back at modelling level. This is of critical importance in order to assist the developer in understanding at a glance the relationships between expected and actual behaviour, without having to inspect the related generated code. Thanks to the automated support, the process can be iterated at will until the developer is satisfied with the results.

Specific Contribution.

Considering that the round-trip support and its usefulness in aiding the preservation of EFPs from models to generated code have been already presented in [5], the specific contributions of this paper are the following:

1. Enhancements to the code generation for entailing the synthesis of both single-process and multi-process configurations (Section 6.1);
2. Specific extensions to the execution platform in order to enable per process monitoring features for EFPs (Section 6.2);
3. Adaptation of the back-propagation facilities (i.e., inplace model transformations) to address newly introduced monitoring features (Section 6.3);
4. Exploitation of the synergy between round-trip support and measurements gathered at system implementation (or runtime) level towards deployment optimisation when developing embedded systems (Section 6.4).

3. RELATED WORK

Attempts that propose to solve the problem by back-propagation, i.e., by reporting the measured values back to the models in order to possibly fix and/or refine estimated numbers can be found in the literature. Navabi et al. in [15] in the early 90’s, and later Mahadevan and Armstrong in [16], proposed different approaches for back-annotating behavioural descriptions with timing information; however, both operate horizontally in terms of abstraction levels and no automation is provided.

Moreover, Varró et al. propose in [17] back-propagation for enabling execution traces retrieved by model checkers or simulation tools to be integrated and re-executed in modelling frameworks; some similarities to our approach might be found when dealing with traceability issues, although the two approaches aim at solving two different problems. An approach similar to ours is described by Guerra et al. in [18] where back-propagation of analysis results to the original model by means of triple graph patterns is described. Nevertheless, the approach is meant to horizontally operate at modelling level with propagation of data among models. 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 without executing the code on a specific platform.

In the literature there exist approaches dealing with deployment optimization based on measurements at system implementation level as described in [19]. A dated approach by Yacoub in [20] introduces systematic measurements of a component-based system, but no tool support is provided.

Horizontal and vertical are used for specifying the direction of data transitions among artefacts; therefore with horizontal we intend transitions from model to model, while with vertical we mean those from code to model.
The COMPAS framework by Mos et al. [21] is a performance monitoring approach for J2EE systems. Components are EJBs and the approach consists of monitoring, modelling, and prediction. An EJB application is augmented with proxy components for each EJB, which send timestamps for EJB life-cycle events to a central dispatcher. Performance measurements are then visualized with a proof-of-concept graphical tool and a modelling technique is used for generating UML models with SPT annotations from the measured performance indices. Then, for performance prediction of the modelled scenarios, the approach suggests using existing simulation techniques, which are not part of the approach. Based on the COMPAS framework, two further approaches have been proposed: AQUA, by Diaconescu et al. in [22], and PAD, by Parsons et al. in [23]. Both approaches expect working J2EE applications as input. AQUA focuses on adapting a component-based application at runtime if performance problems occur. The main idea is that a software component (EJB) with performance problems is replaced with one which is functionally equivalent from a set of redundant components. Furthermore, the approach involves monitoring the workload of a running application. PAD focuses instead on automatic detection of performance anti-patterns in running component-based systems. The approach targets EJB systems and includes performance monitoring, reconstruction of a design model, and anti-pattern detection.

The described approaches assume that a complete component-based system has been implemented and can be run. The goal is therefore to identify performance problems in the running system and adapt the implementation to make it able to fulfil EFPs requirements. Instead, the uniqueness of our round-trip approach consists in introducing a new dimension to deployment optimization at model level with the help of measurements gathered at system implementation level. In fact, when measurements are completed, the code is not manually tuned, but changes to the system are rather performed at model level from which code is re-generated. Doing so, consistency between models and code is kept and the validity of decisions made at model level is likely to be preserved at code level (and the other way around). Moreover, by exploiting the accuracy of system implementation level measurements at modelling level, the developer is relieved from complex code inspection and error-prone manual tuning of code.

Regarding measurements of EFPs at system implementation level, besides runtime monitoring, other verification techniques (e.g., static analysis) can be used for small and simple systems, but their application for large and complex systems might not always be practical and economical [25]. Even in cases where such techniques are feasible, conditions that cause invalidation of the analysis results at runtime may happen. An example of such is the difference between the ideal execution environment (considered for performing analysis) and the actual one which leads to the violation of the assumptions that were taken into account when performing static analysis [26]. Therefore, the information gathered through monitoring the execution of a system is not only interesting and useful for observing the actual behaviour and to detect violations at runtime, but also to be used for making adaptation decisions, as well as to induct enforcement and preservation of properties. Our work in [27] serves as an example of using monitoring information for balancing timing and security properties in embedded real-time systems. Also in [6], we provided an approach for improved enforcement and preservation of timing properties in embedded real-time systems. Hulseius and Andersson in [28] present a method for the synthesis of models of embedded real-time systems from the monitoring information collected from their execution. In this paper, however, we exploit monitoring results, from which observed values are extracted and used to refine design models with EFPs’ values detected during the execution of the system.

4. THE AAL2 SUBSYSTEM: A RUNNING EXAMPLE

The solution proposed in this work has been validated against industrial case-studies modelled in CHESS-ML as described in Section 7. In order to show a concrete application of the proposed solution, we employ the Asynchronous Transfer Mode (ATM) Adaptation Layer 2 (AAL2) subsystem, originally intended to adapt voice for transmission over ATM and currently used in telecommunications as part of connectivity platform systems. The AAL2 subsystem is composed by several thousands of component instances and multiple levels of hierarchical composition of components. Due to its verbosity and complexity, we will exploit a simplified version of the AAL2 to show how the approach operates. Nevertheless, the validation of the approach has been carried out exploiting the complete AAL2 subsystem. In Fig. 1 we propose the simplified version of the AAL2 subsystem (i.e., SwSystem composite component) which is composed by three main components: (i) NCC_i, (ii) AAL2RI_Client_i, (iii) NCIClient_i. Note that the multiplicity (in terms of number of instances) of components and ports is depicted within square brackets. Each of the components has a complex internal structure in terms of composition of other components; in this example we consider only part of the NCC_i internal structure while considering AAL2RI_Client_i and NCIClient_i as stubbed. NCC_i is a connections handler providing connectivity services for the establishment/release of communication paths between pairs of connection endpoints handled by AAL2RI_Client_i. NCIClient_i represents an application asking for services provided by NCC_i and its underlying layers; the components communicate through functional interfaces (function calls or message passing depending on the deployment configuration) exposed by their provided ports.

The NCC_ci is a composite component (Fig. 1), and in this study we focus on: NodeConnHandler_i, which dispatches the incoming connection requests to available NetConn_i instances, NetConn_i, that controls establishment and release of network connections between nodes (NodeConnElem_i instances), NodeConnElem_i, that handles management of connections to the network within the single node, and PortHandler_i, which manages connection resources. Each of these subcomponents has in turn a complex internal structure in terms of components composition; in this case-study we consider only the first two levels of decomposition (down to the NCC_ci’s internal structure).

The behaviour of the system (NetConn_i state-machine is defined through state-machines and are enriched with action code definitions for the involved operations specified by means of ALF in order to reach the required level of expressive power needed to automatically generate full code. Com-
components are connected by means of ports and links between them. The communication is thereby performed by calling operations on the component’s required ports that propagate the invocation to the component owning the provided ports connected to them (note that connected provided and required ports share the same Interface).

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

5. THE ROUND-TRIP SUPPORT

This work proposes the exploitation of a round-trip technique to provide an automated support to aid deployment decisions at modelling level in the development of component-based embedded systems. Such a support is achieved by exploiting the multiple benefits of the round-trip technique, namely the generation of full-fledged functional code from source models, a set of monitoring features to gather values of EFPs of interest from the execution of generated code on the target platform, and the possibility to automatically propagate the computed values back to the source models. Doing so, source models can be evaluated and possibly tuned in terms of deployment of software components on processes to enable a better resource utilisation based on actual values.

In order to maintain consistency between different artefacts in the automated process and therefore ensure correctness of monitoring and back-propagation activities, the generated code is not meant to be edited by hand. Possible optimizations are indeed not performed directly through code editing, but rather by re-iterating the code generation process once the deployment model has been refined according to the evaluation of the back-propagated EFPs’ values. This is especially important in the context of complex systems and in large organisations, where usually code tweaks are done at sub-system and sub-department levels to achieve, for instance, better performance. This can in turn lead to inconsistencies among development artefacts and modifications often remain unknown to developers at upper levels of the organisational hierarchy [29]. The fact that EFPs (e.g., memory usage and performance) are not independent and have inter-dependencies further emphasises the importance of having such a consistency mechanism for development artefacts since, most often, EFPs cannot be considered in isolation and their mutual impacts and trade-offs need to be taken into account too [30]. The general goal of this work is to demonstrate that the round-trip support helps the developer in determining the most suitable deployment configuration based on actual values gathered by executing and monitoring code which is automatically generated from source models (Fig. 2). Once design modelling tasks have been successfully completed, the objective is to enable automatic generation of implementation code from source models. Taking design models as source arte-
6. FROM MODELS TO CODE AND BACK

In this section we show the application of the round-trip support to the AAL2 subsystem in terms of generation of C++ code from the source model described in Section 4, monitoring of the code execution on OSE and back-propagation of gathered values to the CHESS model. Moreover, after performing deployment tuning based on such values, we re-iterate the process to show how these modifications at modelling level can affect EFPs of re-generated code.

6.1 Generation of C++ from CHESS Model

A set of model-to-model transformations operates on the CHESS model to generate three different intermediate artefacts: i) Instance model, that stores component instances and links among them via ports [32], ii) Intermediate model, that represents the main intermediate artefact in the process and contains all the information needed to generate code [5], and iii) Back-propagation model, that contains explicit traceability links between model elements and code as well as place-holders for EFPs values coming from the following monitoring activities [5]. The intermediate model is enriched with behavioural information defined in ALF by means of parsing operations and in-place model-to-model transformations. At this point, the C++ implementation code is generated through model-to-text transformations taking as input the intermediate model [5].

Each of these intermediate artefacts conforms to its corresponding metamodel that we expressly defined for the code generation process. With regards to the technologies employed in the generation process, model-to-model transformations are defined by means of Operational QVT, while model-to-text transformations are defined in Xpand [33].

As aforementioned, the synthesis described in [5] was limited to single-process applications, which prevented its employment for deployment issues. The approach has been therefore enhanced in order to be able to consider deployment configurations specified at modelling level and therefore generate single- or multi-process implementation accordingly. Deployment is described in CHESS-ML by means of an ad-hoc model in which the modeller defines allocation of components instances to processes with the help of specific concepts provided by MARTE. In Figure 3 a portion of the deployment model concerning the AAL2 subsystem is depicted. More specifically the following possible allocations are shown:

- Different instances of a same component on different processes: represented by the two instances of the component NCIClient_i (SwSystem_NCIClient_inst) respectively allocated to processA and processB (Figure 3.a);
- Single component instance on a process: represented by the instance of AAL2RClient_i (SwSystem_AAL2RClient_inst) allocated to processC (Figure 3.b);
- Multiple set of component instances on a single process: represented by the entire set of NetConn_i (SwSys-
In order to achieve this, the transformation process proposed has been enhanced to take into account the information carried by the deployment model that, together with the functional model, drives the code generation. More specifically, in the case of code tailored for OSE, that information is exploited to create processes and deploy component instances on them, as well as generate the communication code in order to distinguish between:

- **Intraprocess communication**: communication between components deployed on a same process is achieved by function calls;
- **Interprocess communication**: components allocated to different processes communicate via signals across processes.

More specifically, the communication between components defined in the CHESS-ML model in terms of ALF as function calls on ports had to be properly translated into appropriate intermediate concepts. Depending on the deployment configuration of the communicating components, a function call in the model is translated into (i) a function call, if the components are deployed on the same process, or (ii) into a message `send` (i.e., OSE signal) in the case of components deployed on different processes. This means that the entailment of different deployment configurations has introduced a higher degree of variability in the semantic interpretation of the design models that depends on the modelled deployment. The ability to correctly interpret such information and thereby generate code that reflects the modelled deployment configuration has been embedded in the model transformations responsible for the code generation.

Due to the complexity of the multi-process code generation, the details related to the involved artefacts ranging from intermediate metamodels to model transformations will be dealt with separately. For the interested reader, preliminary information about the single-process generation chain can be found in [5].

### 6.2 Monitoring Code Execution

In order to enable the needed monitoring features, specific extensions to the execution platform have been made. These extensions have been implemented mainly in the form of two additional system processes: one for monitoring and another for logging. These two processes are assigned lower priorities than the generated application ones. The monitoring process is responsible for calculating and determining the values for EFFs of interest for both the whole system (e.g., overall CPU usage) as well as per process. The actual task of logging such information is separated from the monitoring process and performed by the logging one. This separation allows to mitigate the side effects of resource-demanding I/O activities. When a request for monitoring is issued by one of the application processes, the monitoring process starts executing and determining EFFs values. The information to be logged is sent to the logging process by the monitoring one through opposite signals. Therefore, if the logging process does not get the needed CPU time to perform its job, the signals sent to it are pushed in its signal queue, maintained automatically by OSE, and processed as soon as it gets to execute.

The implemented monitoring process is capable of determining values for the following properties:

- System-level properties: total CPU load, total number of generated signals in the system, system throughput (sent and received packets), total number of processes in the system;
- Process-level properties: total execution time of a process (from the startup of the system including all invocations of it), instance execution time of a process (one invocation and instance), response time of a process, heap and stack usage for a process, number of signals generated by a process, and CPU load of a process.

To calculate execution and response times, `swap_in` and `swap_out` handlers of OSE have been used. The former event handler is invoked each time a process gets CPU to execute, and the latter is invoked when CPU is taken from a process and it is preempted. The algorithms and mechanisms for the calculation of execution and response times have been implemented into these two event handlers. However, since they are invoked for every process in the system, additional tweaks were made in order to filter their executions for only the generated application processes which are of interest.

The monitoring activities give a textual description of the gathered values as output. The results of monitoring the execution of the C++ code generated from the AAL2 model and with the deployment configuration partially depicted in Fig. 3 are shown partially in Listing 1.

#### Listing 1: Monitored Properties

<table>
<thead>
<tr>
<th>Time</th>
<th>Process ID</th>
<th>Monitored Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>466</td>
<td>mprocA_0003c</td>
<td>S_CPU_LOAD</td>
<td>29.978</td>
</tr>
<tr>
<td>467</td>
<td>S_NUMBER_OF_PROCESSES</td>
<td>73476</td>
<td></td>
</tr>
<tr>
<td>468</td>
<td>S_NUMBER_OF_SIGNALS</td>
<td>365</td>
<td></td>
</tr>
<tr>
<td>469</td>
<td>MMTBF</td>
<td>558,617747</td>
<td></td>
</tr>
<tr>
<td>470</td>
<td>S_THROUGHPUT</td>
<td>942.1676</td>
<td></td>
</tr>
<tr>
<td>471</td>
<td>P_HEAP_USAGE</td>
<td>384.512</td>
<td></td>
</tr>
<tr>
<td>472</td>
<td>P_STACK_USAGE</td>
<td>1536.0.2048</td>
<td></td>
</tr>
<tr>
<td>473</td>
<td>P_NUMBER_OF_SIGNALS</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>474</td>
<td>P_EXECUTION_TIME</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>475</td>
<td>P_RESPONSE_TIME</td>
<td>1609984</td>
<td></td>
</tr>
</tbody>
</table>

The first column in Listing 1 indicates the time instance at which monitoring has been performed (in system ticks unit). The second column identifies the type of the monitored information; the properties beginning with ‘S’ indicate system-level values while the ones starting with ‘P’ identify a process-level value (e.g., S_NUMBER_OF_SIGNALS: total number of signals in the system at the moment of monitoring, P_NUMBER_OF_SIGNALS: total number of signals owned by a process). The values after the name of the process (i.e., mprocA) indicates process’ ID in hexadecimal and decimal format respectively. As it can be seen, some of the properties have multiple values, in which case they mean different aspects related to the same property. For
example, the first value related to P_HEAP_USAGE represents the heap size requested by the process and the second one shows the actual heap size allocated for the process by the operating system (the difference between the two is due to factors such as memory paging and memory management mechanisms of OSE).

### 6.3 Back-propagation to CHESS Model

While a similar approach had been proposed in [5], since the monitoring features, and thereby the output format, had entirely changed, the back-propagation facilities had to be consequently adapted. More specifically, a specific in-place text-to-model transformation has been implemented from scratch in order for the monitoring results of interest to be manipulated and stored in the back-propagation 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. The actual enrichment of the CHESS model with such EFPs values is performed through a model-to-model transformation. Taking as input the CHESS model and the back-propagation model, it performs a set of in-place transformations (adapted version of the ones proposed in [5]) on the CHESS model to enrich it with the observed values stored in the back-propagation model. The results of the back-propagation are partially shown by means of extra-functional decorations of the AAL2 model in Fig. 4. More specifically, we can notice that values concerning mprocA are back-propagated to instance 1 of NCI_Client_i (marked by [1] – e.g., EXECUTION_TIME[1]) while the ones carried by mprocC apply to the single instance of AAL2RI_Client_i. Such correspondence is stored in the back-propagation model and originates from the deployment model depicted in Fig. 3. Regarding the technologies used in the back-propagation process, model-to-model transformations are defined by means of Operational QVT as well as text-to-model transformations (which are aided by structured text parsers defined in Java). Once the monitored results have been back-propagated, the developer has at her disposal the modelled system enriched with actual values gathered at runtime. The values depicted in Listing 1 and propagated back to the AAL2 model (in Fig. 4) are related to the deployment configuration in which the component instance NCI_Client_i[1] is deployed on ProcessA (mprocA in Listing 1) and the component instance AAL2RI_Client_i[1] is allocated to ProcessC (mprocC).

### 6.4 Evaluation of Deployment Configurations

At this point, let us try out a different deployment configuration in which we allocate both NCI_Client_i[1] and AAL2RI_Client_i[1] to ProcessA, since the communication between them is quite dense. Once the model is modified, the code can be regenerated and its execution monitored. In Table 1 the monitoring results concerning both the first as well as the tuned deployment configurations are depicted. Configuration 1 represents the deployment of the two component instances on separate processes, while 2 concerns the deployment of both instances on a single process.

As we can see, by changing the deployment configuration, in the specific case by allocating the two instances on the same process, we actually experience a decrease of the execution time of AAL2RI_Client_i from 11 to 3. Besides this reduction which may not be relevant in the actual employment of the system, what we aimed at pointing out was the usefulness of having an automatic mechanism for gathering and back-propagating runtime values to model level for allowing thorough evaluation of the system’s deployment configuration. While providing meaningful values at model level, the approach is not yet able to provide any hint on how to interpret them. Limitations of the current solution in this sense as well a future enhancements towards further automation in the tuning phase are discussed in the next section.

<table>
<thead>
<tr>
<th>Config.</th>
<th>Comp. Instance</th>
<th>Process</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NCI_Client_i[1]</td>
<td>A</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>AAL2RI_Client_i</td>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>NCI_Client_i[1]</td>
<td>A</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>AAL2RI_Client_i</td>
<td>A</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: Different deployments that induce different monitoring results

### 7. DISCUSSION AND FUTURE WORK

Validating the approach against an industrial case-study gave us the possibility to evaluate the scalability of the proposed solution by testing several design model sizes for the same system. The hardware configuration was composed by a Windows machine running on a 2.6GHz CPU and 8GB RAM. Moreover, we employed the OSE Soft-Kernel (SFK) 5.5.1 as target platform for code execution. The SFK edition provides a simulated environment of the actual target embedded board on a host machine (a Windows or Linux host) and accurately reproduces the behaviour of the real environment. Focusing on the most time-consuming task in the approach (i.e., automatic code generation) and considering n as the greatest number of instances per component, m as the greatest number of instances per port, and k as the number of hierarchical composition level, the general limit behaviour of the computation is represented by $O((n+m)^k)$. Overall the proposed solution, on a model with $k = 2$, resulted very scalable up to $n + m = 10^3$ (i.e., within 5 minutes) while gradually slowing down for greater number of instances (e.g., over 30 minutes for $n + m = 10^4$). More specifically, in terms of number of component and port in-
stances (i), the generation time (in seconds) was: (i) 52s for \( i = 90 \), 113s for \( i = 1853 \), and 831s for \( i = 16003 \) (complete AAL2 subsystem). However, the presented validation has been performed on a preliminary implementation of the code generator which focuses mostly on the correctness of the generation process with no major emphasis on performance issues. That is why we expect to achieve better results, in terms of scalability, with a more mature version of the code generator.

Regarding the time needed for monitoring activities, it heavily depends on the duration of the code execution since the measurements themselves are mostly performed by parallel processes during the execution. Nevertheless, the calculation of more complex EFPs could introduce additional complexity hence requiring additional computation time.

Concerning back-propagation tasks, they resulted to be more scalable thanks to the detailed information, concerning the path to the specific model element to be annotated, carried by the back-propagation model. This means that most of the needed computation is performed when generating the back-propagation model, while the actual values injection, first from monitoring results to back-propagation model and therefore from the latter to the design model only involves an update of specific values with no need of complex searches nor navigations.

Generally, the number of iterations for reaching the desired EFPs depends on the accuracy of measurements as well as the modeller’s ability in both modelling the system and also effectively employ the back-propagated values to tune the models accordingly. That is to say that the developer is supposed to be able to understand the back-propagated values in relation to the expected behaviour and thereby tune the models accordingly to generate a better-performing implementation. In aid to the modeller, model-based analysis and deployment optimization techniques could be exploited to minimise the number of iterations.

As described in the previous sections, measurements of EFPs are generally performed at process level, which means that in order to obtain component-level values, each component should be deployed on a separate process. This, although representing an important step forward in comparison to the previous version of our approach, only opens up to further enhancements. In fact, while in principle the back propagation can handle the case of several components mapped to one process, the monitoring facilities cannot due to the execution platform. Therefore, future improvements will entail the possibility to always monitor properties at runtime per component even when several components would be deployed on a same process. In order to achieve this, both improved dedicated marking in the generated code as well as specific tweaks at platform level should be defined and implemented.

We plan to further enhance the code generation in order to address deployment on multiple processing units, which would amplify the benefits of multiprocessing addressed in this work (i.e., multi-process on single processing unit).

As aforementioned, starting from an initial model, the approach is iterated at will until the EFPs requirements are fulfilled. Therefore, once providing support for multicore solutions, there would be the possibility to minimise the number of such iterations by exploiting model-based deployment optimization techniques (e.g., [34] by Feljan et al.) to enhance the deployment-tuning phase with a semi-automated approach that combines both runtime measurements as well as design-time performance predictions.

Additionally, as for the monitoring part, by adding the support for further EFPs, it becomes possible to take into consideration more parameters for tuning the deployment of the system. It would be possible, as an extension of the monitoring part, to enable the calculation of MTBF for each generated process, and therefore for the components that are mapped to it. Having MTBF values as one of the parameters indicating the dependability of the system, would add a further dimension (i.e., dependability) at model level that may help in making deployment decisions. Similarly, if there would be a mechanism to determine an indicator value for the energy consumption of each process (e.g., something similar to the ACPI standard [35]) it would become possible to even tune the system in terms of energy consumption. The feasibility of this possible enhancement needs further investigations.

8. CONCLUSION

In this paper we described a possible solution to relieve the developer’s effort in optimising deployment configuration when developing component-based embedded systems with the help of accurate measurements gathered at system implementation level and avoiding manual editing of code. The solution consists of a fully automated model-driven round-trip approach that enables: (i) generation of full-fledged code from source models entailing customisable single- and multi-process deployment configurations, (ii) monitoring of code execution for measuring a selected set of EFPs at system implementation level, and (iii) back-propagation facilities to enrich the source models with such measurements.

By operating exclusively at modelling level, the developer is discharged from inspecting as well as editing generated code. At the same time, through automatic back-propagation of measurements gathered at system implementation level, the approach assists the developer in understanding at first sight the relationships between expected and actual behaviour hence enabling a well-aware tuning of deployment configuration at model level.

While a previous version of the round-trip support (i.e., only single-process deployment was entailed) had been already described in [5], in this work we describe how we enhanced it to enable synthesis of design models to generate either highly resource efficient (in terms of inter-system communications) single-process applications or exploit multi-process configurations that could run in parallel on multicore. Moreover, having at our disposal this variety of deployment options, we were able to focus on how the round-trip support can employ measurements at system implementation level towards deployment optimization and we showed it by applying the solution to an industrial example in the telecom-domain. Additionally, we described complexity and scalability of the different steps constituting the solution, as well as interesting directions for possible future enhancements.

Acknowledgments

This research is supported by the RALF3 (Swedish Foundation for Strategic Research (SSF), http://www.mrtc.mdh.se/projects/ralf3/) project and the Swedish Knowledge Foundation (KKS) through the ITS-EASY research school.
9. REFERENCES


