Exploring Timing Model Extractions at EAST-ADL Design-level using Model Transformations

Alessio Bucaioni*†, Saad Mubeen*†, Antonio Cicchetti* and Mikael Sjödin*
* Mälardalen Real-Time Research Centre (MRTC), Mälardalen University, Västerås, Sweden
† Arcticus Systems AB, Järfälla, Sweden
*{alessio.bucaioni, saad.mubeen, antonio.cicchetti, mikael.sjodin}@mdh.se
†{alessio.bucaioni, saad.mubeen}@arcticus-systems.com

***Abstract*—We discuss the problem of extracting control and data flows from vehicular distributed embedded systems at higher abstraction levels during their development. Unambiguous extraction of control and data flows is vital part of the end-to-end timing model which is used as input by the end-to-end timing analysis engines. The goal is to support end-to-end timing analysis at higher abstraction levels. In order to address the problem, we propose a two-phase methodology that exploits the principles of Model Driven Engineering and Component Based Software Engineering. Using this methodology, the software architecture at a higher level is automatically transformed to all legal implementation-level models. The end-to-end timing analysis is performed on each generated implementation-level model and the analysis results are fed back to the design-level model. This activity supports design space exploration, model refinement and/or remodeling at higher abstraction levels for tuning the timing behavior of the system.**

I. INTRODUCTION

The intrinsic complexity of vehicular embedded systems demands for development methodologies and technologies that are able to cope with it. In the last decades, Component-Based Software Engineering (CBSE) [1], [2], Model-Driven Engineering (MDE) [3] and their crosplay have gained acceptance due to their ability to both reduce the development complexity, by raising the abstraction level, and to cope with the most arduous aspects of these systems such as timing and safety requirements [2].

EAST-ADL [4] together with its development methodology has been getting closer and closer towards the status of de-facto standard within the automotive domain. It defines a top down development process promoting the separation of concerns through a four-level architecture, where each level is designed for hiding details pertaining to higher or lower levels. At the lowest level, i.e., implementation level, EAST-ADL makes use of AUTOSAR [5], which is an industrial initiative to provide standardized software architecture for the development of vehicular embedded systems. While EAST-ADL methodology has been successful in raising the software development abstraction level, it provides few means for coping with the timing requirements of such software systems. In the past few years, several initiatives such as TIMMO [6] and TIMMO2USE [7] and their outcomes including TADL [8] and TADL2 [9] languages, tried to provide AUTOSAR with a timing model. Nevertheless, they did not fully succeed with this goal at various abstraction levels because AUTOSAR explicitly hides some implementation-level information which is necessary for building a timing model from the software architecture.

Nowadays, automotive industry needs development methodologies and technologies able to cope with the timing requirements of such software systems. Nevertheless, current industrial needs push for having such end-to-end timing analysis earlier during the development process, i.e., at the design level. Industry is currently reusing most of the software architecture from previous projects, that means, some crude software architecture is already available in the early stages of the software development. In this context, it is beneficial to perform early timing analysis for Design Space Exploration (DSE) and software architecture refinements.

A. Paper Contribution

We target core challenges that are faced when end-to-end timing models are extracted to support end-to-end timing analysis at higher abstraction levels and earlier stages of the software development of vehicular distributed embedded systems. These challenges include extraction of data and control paths at the implementation level from the design-level models; transformation of multiple implementation-level models from a single design-level model; and dealing with these transformed models from the timing analysis point of view. In order to deal with these challenges, we propose a two-phase methodology that exploits the principles of MDE and CBSE. In the first phase, the software architecture of the system at the EAST-ADL design level is automatically transformed to all legal implementation-level models, e.g., models that are build using the Rubus Component Model (RCM) [10]. Whereas in the second phase, the end-to-end timing analyses are performed on each generated implementation-level model. The analysis results of all or selected implementation-level models are fed back to the design-level model. Thus, the methodology provides a support for DSE and models refinement. Moreover, it supports remodeling at higher abstraction levels for the purpose of tuning the timing behavior of the system.

B. Relation with Authors' Previous Works

In [11], we provide a method to extract timing models and perform end-to-end timing analysis of component-based distributed embedded systems. In [12], RCM is presented as an alternative to AUTOSAR in the EAST-ADL development methodology and its usage is discussed for enabling end-to-end timing analysis at the lowest EAST-ADL abstraction level, i.e., implementation level. In [13], RCM is extended with a concrete meta-model definition. In [14], the translation of timing constraints from the design- to the implementation-level models is provided. However, the translation is done manually and is limited by the constraint such that it only considers that implementation-level model which results in worst-case response times and delays. In comparison with above works, this paper presents a novel two-phase methodology to automatically transform the software architecture of the system at the EAST-ADL design level to all legal implementation-level models (RCM models). The existing analysis engines in the Rubus analysis framework perform timing analysis on each generated implementation-level model. The analysis results are then fed back to the design-level model to support DSE and models refinement.

II. Background and Related Works

A. EAST-ADL Development Methodology

EAST-ADL defines a top-down development methodology that promotes the separation of concerns through the usage of four different abstraction levels, where each level provides a complete definition of the system under development for a specific perspective. Figure 1 shows the abstraction levels architecture together with the methodologies, models and languages used at each level.

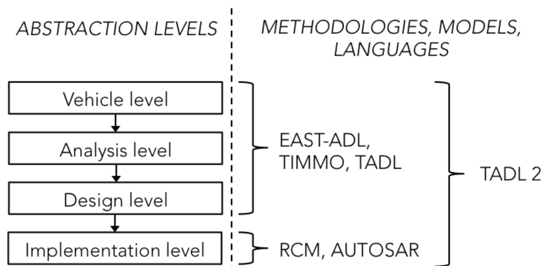

Fig. 1: EAST-ADL abstraction levels

1) Vehicle level: The vehicle level, also known as end-to-end level, serves for capturing all the information regarding what the system is supposed to do, i.e., requirements and features on the end-to-end functionality of the vehicle. Feature models and requirements can be used for showing what the system provides and, eventually, how the product line is organized in terms of available assets.

2) Analysis level: At this level, the end-to-end functionalities are expressed using formal notations. Behaviors and interfaces are specified for each functionality. Yet, design and implementation details are omitted. At this stage, high-level analysis for functional verification can be performed.

3) Design level: At this level, the analysis-level artifacts are refined with more design-oriented details. The architecture of the system is redefined in terms of software, hardware and middleware architectures. Also, software functions to hardware allocation is expressed.

4) Implementation level: The design-level artifacts are enriched with implementation details. Component models are used to model the system in terms of components and their interconnections. The code for vehicle functions can be synthesized from the software component architecture.

B. The Rubus Component Model (RCM)

Rubus [1] is a collection of methods, theories and tools for model- and component-based development of resource-constrained embedded real-time systems. It is developed by Arcticus Systems in collaboration with Mälardalen University. Rubus is mainly used for development of vehicles control functionality by several international companies. The Rubus concept comprises of RCM and its development environment Rubus-ICE (Integrated Component development Environment), which includes modeling tools, code generators, analysis tools and run-time infrastructure. RCM has been recently extended with a concrete meta-model definition [13] for embracing the MDE vision and streamlining the modeling language.

RCM is used for expressing the software architecture in terms of software components and interconnections. A software component in RCM is called Software Circuit (SWC) and represents the lowest-level hierarchical element. Its purpose is to encapsulate basic functions. RCM distinguishes the SWCs interactions by separating the data flow from the control flow. The latter is defined by triggering objects, i.e., clocks and events. SWCs communicate with each other via data ports. RCM facilitates analysis and reuse of components in different contexts by separating functional code from the infrastructure that implements the execution environment. Within the context of above-mentioned abstraction levels in Figure 1, RCM is used at the implementation level.

C. End-to-end Timing Models and Analyses

An end-to-end timing model consists of timing properties, requirements, dependencies and linking information of all tasks, messages and task chains in the distributed embedded system under analysis[2]. It can be divided into timing and linking models. For instance, consider a task chain distributed over three nodes connected by a network as shown in Figure 2. The system timing model contains all the timing information about the three nodes and the network. Whereas the system linking model contains all the linking information in the task chains, including the control and data paths.

The analysis engines [11] use these models for performing end-to-end timing analyses. The analyses results include response-time of tasks and messages as well as system utilization. Also, the analysis engines calculate the end-to-end

[1]http://www.arcticus-systems.com

[2]We refer the reader to [12] for details about the timing model.

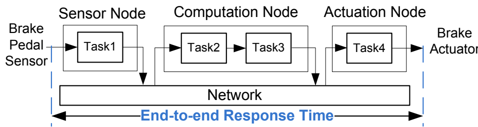

Fig. 2: Example demonstrating end-to-end response time

response times and delays. The end-to-end response time of a task chain is equal to the elapsed time between the arrival of an event, e.g., the brake pedal sensor input in the sensor node and the response time of task, e.g., the brake actuation signal in the actuation node as shown in Figure 2.

Within a task chain, if the tasks are triggered by independent sources, then it is important to calculate different types of delays such as age and reaction. Such delays are crucial in control systems and body electronics domains, respectively. An age delay corresponds to the freshness of data, while the reaction delay corresponds to the first reaction for a given stimulus. In order to explain the meaning of reaction and age delays, consider a task chain in a single-node system as shown in Figure 3. There are two tasks in the chain denoted by τ_1 and τ_2 and triggered by independent clocks of periods 25ms and 5ms respectively. Let the Worst-Case Execution Times (WCETs) of these tasks be 2ms and 1ms respectively. τ_1 reads data from the register Reg-1 and writes data to Reg-2. Similarly, τ_2 reads data from the Reg-2 and writes data to Reg-3. Since, the tasks are activated independently with different clocks, there can be multiple outputs (Reg-3) corresponding to one input (Reg-1) to the chain as shown by several uni-directional arrows in Figure 4. The age and reaction delays are also identified in the figure. These delays are equally important in distributed embedded systems.

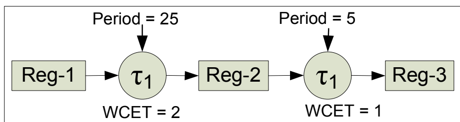

Fig. 3: A task chain with independent activations of tasks

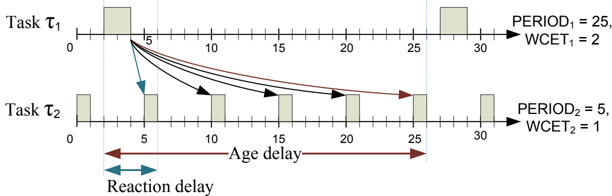

Fig. 4: Example demonstrating end-to-end delays

D. Model Driven Engineering (MDE) and Janus Transformation Language (JTL)

MDE is a discipline which aims to abstract software development from the implementation technology by shifting the focus from the coding to the modeling phase. In this context, MDE promotes models and model transformations as first-class citizens. Models are seen as an abstraction of a real systems, built for a specific purpose [3]. Whereas, model transformations can be seen as a gluing mechanism among models [15]. Rules and constraints for the models' construction are specified in the so-called metamodels, i.e., a language definition to which a correct model must conform.

JTL [16] is a declarative model transformation language tailored to support bidirectionality and change propagation. The JTL transformation engine is implemented by means of the Answer Set Programming (ASP) [17], that is a form of declarative programming oriented towards difficult search problems and based on the stable model (answer set) semantics of logic programming. In JTL, a model transformation between a source and a target model, is specified as a set of relations among models, which must hold for the transformation to be successful. The transformation engine considers such mapping rules for generating the set of all possible solutions. Then, it can refine the generated set by applying constraints on the generated target models, i.e., meta-model conformance rules.

E. MDE for DSE

During the last decades, MDE has been successfully employed for DSE. In [18], the author exploit JTL for implementing an automatic deployment exploration technique based on refinement transformations and platform-based design. The technique is validated upon an automotive case study using an AUTOSAR-like metamodel. [19] presents a pattern catalog for categorizing different MDE approaches for DSE. It demonstrates the usage of the identified patterns with a literature survey. The work in [20] defines a guided DSE approach based on selection and cut-off criteria defined using dependency analysis of transformation rules and an algebraic abstraction. Cutt-off criteria are used to identify dead-end states, while selection criteria are used to order applicable rules in a given state. The methodology has been effectively evaluated upon a cloud configuration problem.

III. Problem Statement

In order to support the end-to-end timing analysis at the design level, the end-to-end timing model should be extracted from the design-level model of the application. Consider the design-level model of a component chain consisting of three software components shown in Figure 5. Among other parameters, complete control (trigger) and data paths along component chains (task chains at run-time) must be unambiguously captured in the timing model. Unambiguous extraction of control and data paths from the system are vital for performing its timing analysis.

A control path captures the flow of triggers along the components chain, e.g., control path of the chain in Figure 6(b) can be expressed as $\{\{\text{Sensor} \rightarrow \text{Controller}\}, \{\text{Actuator}\}\}$. This means that Controller SWC is triggered by Sensor SWC, while Actuator SWC is triggered independently. Similarly, control paths of the chains shown in Figure 6(a) and Figure 6(c) can be expressed as $\{\{\text{Sensor} \rightarrow \text{Controller} \rightarrow \text{Actuator}\}\}$ and $\{\{\text{Sensor}\}, \{\text{Controller}\}, \{\text{Actuator}\}\}$ respectively. It should be noted that the three component chains shown in Figure 6

are modeled at the implementation level using the Rubus-ICE tool suite.

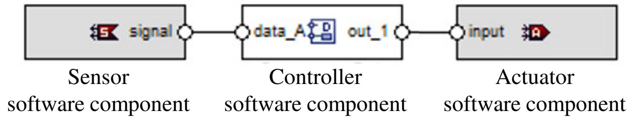

Fig. 5: Design-level model of a component chain

The main challenge faced during the extraction of end-to-end timing models at the design level is the lack of clear separation between control and data paths. Although TADL2 augments EAST-ADL with some timing information at the design level, the support for clear separation and unambiguous extraction of control and data flows is still missing. At the implementation level, e.g. in RCM, these paths are clearly separated from each other by means of trigger and data ports as shown in Figure 6. A trigger output port of an SWC can only be connected to the trigger input port(s) of other SWC(s). Similarly, a data output port of an SWC can only be connected to the data input port(s) of other SWC(s). Hence, the trigger and data paths can be clearly identified and extracted in the timing model. Whereas at the design level, the components communicate by means of *flow ports* as shown in Figure 5. A flow port is an EAST-ADL object that is used to transfer data between components. It has a single buffer. The data contained in the port is non-consumable and over-writable. Since there is no other explicit information available about this object, it can be interpreted as a data or a trigger port at the implementation level. There is no support to specify explicit trigger paths at the design level. Moreover, a component can be triggered via specified timing constraints on events, modes, or internal behavior of the component. For example, consider again the design-level model of a component chain shown in Figure 5. Assume there is a periodic constraint of 10ms specified on this chain. There can be three model interpretations of this chain at the implementation level as shown in Figure 6. Consequently, there are three different control flows in these models. The data flow and control flow should be clearly and separately captured in the end-to-end timing model because the type of the timing analysis depends upon it. For example, it is not meaningful to perform end-to-end delay analysis on a trigger chain as shown in Figure 6(a) [11].

We have considered a very small part of a large system in the above example. In reality, distributed embedded systems may contain hundreds of software components and component chains. The component chains, in turn, may be distributed over several nodes or Electronic Control Units (ECUs). Intuitively, there can be a large number of implementation-level model interpretations of the design-level model of a single distributed chain. To the best of our knowledge, RCM is the only model that intends to support high-precision end-to-end timing analysis at the design level[3]. However, it considers only that implementation-level model interpretation of the design-level

[3]The solution is being prototyped.

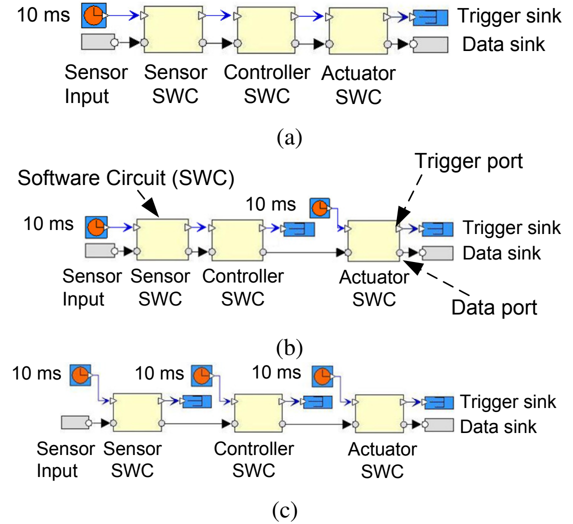

Fig. 6: Implementation-level models of the design-level model of the component chain in Figure 5

model which produces worst-case response-times and delays. As a result, the calculated response-times and delays may be very pessimistic (considerably large compared to actual response times and delays). In order to be less pessimistic with the analysis results, the end-to-end timing analysis should be performed on all possible implementation-level model interpretations of a design-level model. The analysis results of all these models should be presented to the user. The user should be able to select the model with respect to the analysis results. This activity also helps in doing DSE and performing model refinements earlier during the development. There is a need for a methodology and corresponding automated model transformations to deal with this problem.

IV. Proposed Solution and Methodology

In order to address the problem discussed in the previous section, we propose a solution methodology as shown in Figure 7. The input to the methodology is the EAST-ADL design-level software architecture of the system under development. Whereas, the output of the methodology consists of the end-to-end timing analysis results that are fed back to the design-level software architecture. The methodology comprises of two major phases (A) transformation phase and (B) timing analysis phase.

A. Transformation phase

The transformation phase is realized as a model-to-model transformation between EAST-ADL design-level and RCM models. The mapping relation between the related metamodels is a non-surjective relation. We select JTL to implement the transformation because it is able to deal with partial information, information loss and uncertainty [16]. To the best of our knowledge, JTL is the only transformation language with such characteristics. The JTL transformation requires the EAST-ADL design-level model and metamodel as well

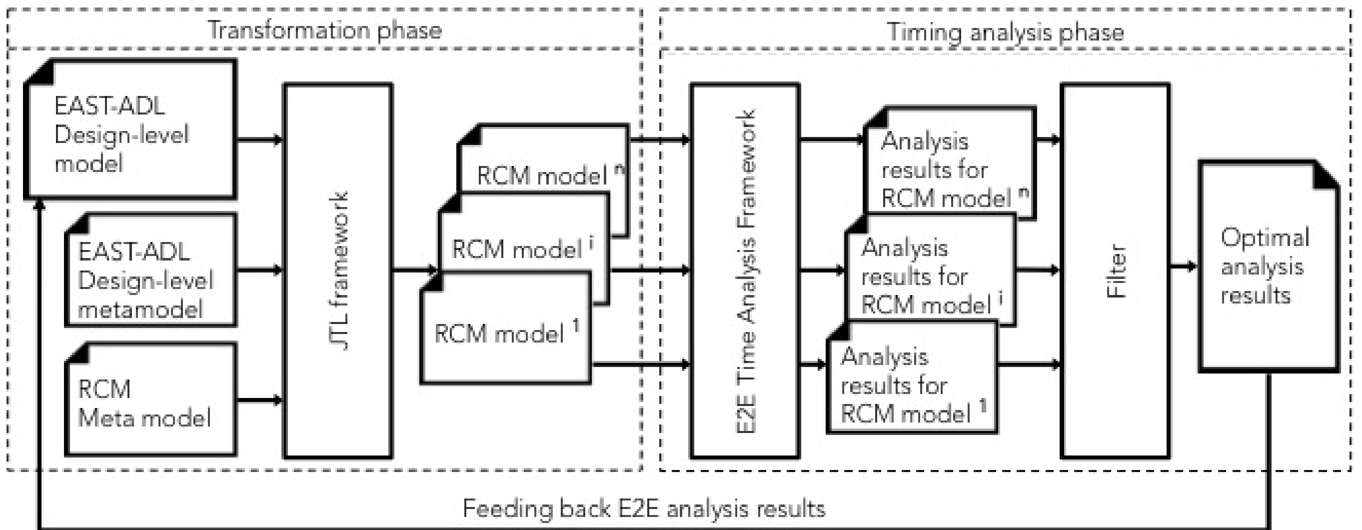

Fig. 7: Methodology of the proposed solution

as the RCM metamodel as inputs. Exploiting the ASP engine, JTL produces, with a single execution, all the possible RCM models for the specified EAST-ADL design-level model. The transformation assumes a one-to-one mapping between each design- and implementation-level component. Although a design-level component can be mapped to more than one implementation-level components, our assumption of one-to-one mapping is based on common practice in industry, especially in the segment of construction-equipment vehicles domain. All the generated implementation-level models have same data flows but different control flows. For instance, consider that the EAST-ADL design-level model shown in Figure 5 along with the EAST-ADL and RCM metamodels are provided as input to the JTL framework. The corresponding transformation results into three implementation-level models as shown in Figure 6. For a complex embedded application, there can be many such transformations of a design-level model.

B. Timing analysis phase

In the timing analysis phase, our methodology exploits the end-to-end timing analysis framework of Rubus-ICE [11]. All the generated implementation-level models from the previous phase are provided as inputs to the analysis framework. It should be noted that the timing analysis framework operates on the implementation-level models which are annotated with complete timing information. However, in the generated models derived from the previous phase, some of the timing information required to do the timing analysis may be missing. In this respect, we make assumptions to compensate for the missing timing information. For example, if worst-, best- and average-case execution times are not specified at the design level, they can be estimated at the implementation level either using estimations by experts, reusing them from other projects or from previous iterations during the model refinement process. Further, we assume that the execution order of design-level components in a chain is specified, otherwise we make implicit assumption about it. That is, each component is assumed to execute only after successful execution of preceding component in the chain, unless specified otherwise. This means, a data provider component is assumed to be always executed before the data receiver component. Since this assumption fixes the execution order, it is safe to assume the priorities of the components are equal within the component chain.

Eventually, the analysis framework performs end-to-end response-time and delay analyses on each implementation-level model separately. Once again, consider the three generated implementation-level models shown in Figure 6. We assume the WCET of each component to be equal to 1ms. Here we are interested in the end-to-end response times, reaction and age delays among all timing analysis results. These times for the three component chains are (a) 3ms, 3ms, 3ms; (b) 3ms, 10ms, 10ms; and (c) 3ms, 29ms, 19ms respectively. These analysis results are provided to the filter module which selects optimal result(s) depending upon the specified constraints (e.g. constraints on timing or constraints on activation of individual components in a chain, i.e., dependent or independent triggering). The filter can be considered as the designer who selects optimal implementation-level model interpretation of the design-level model based on the analysis results. The filter can also be a logical block making such decisions based on the specified constraints (the process of automating the filter is a future work).

The translation from the design- to the implementation-level models is automatic. Moreover, the translation is not limited by the constraint of considering that implementation-level model which results in worst-case timing behavior. For example, in the case of constrained translation, the design-level model in Figure 5 is only translated to implementation-level model of Figure 6 (c) because that chain results in worst-case delays. On the other hand, the timing analysis phase in our current

methodology provides all possible implementation-level model interpretations of the design-level model. For example, the filter module can select the chain in Figure 6(a) or Figure 6(b) as optimal because of lower end-to-end delays and provide the corresponding analysis results back to the design level. Based on this feedback, better decisions can be made during DSE or the refinement of the system model. Moreover, the system can be remodeled or decisions can be made such that the timing analysis results in the next iteration are less pessimistic. This can help in fine tuning the timing behavior of the system.

C. Proof of concept

As a proof of concept we instantiate the above presented methodology within Rubus-ICE as depicted in Figure 8. In Rubus-ICE tool suite, Rubus-EAST tool supports modeling of applications with EAST-ADL. There are two options to start the modeling at the design level: i) model directly in Rubus-EAST, or ii) import XMI formats of EAST-ADL models of the application from any other EAST-ADL designer. The transformation phase of the methodology can be implemented as a plug-in for Rubus-ICE denoted as DL-JTL plug-in, where DL stands for Design Level. According to the proposed methodology, this plug-in calls the JTL framework that generates all feasible RCM models corresponding to the design-level model and provides them back to the plug-in. Consequently, the DL-JTL plug-in calls the HRTA and E2EDA plug-ins [11] and provides all generated implementation-level models to them. The HRTA and E2EDA plug-ins, in turn, perform end-to-end response time and delay analyses of all the input models and provide their analysis results back to the DL-JTL plug-in. Finally, the DL-JTL plug-in selects the optimal analysis results and feeds them back to the design-level model of the application in the Rubus-EAST tool. The sequence of above mentioned steps are identified in Figure 8.

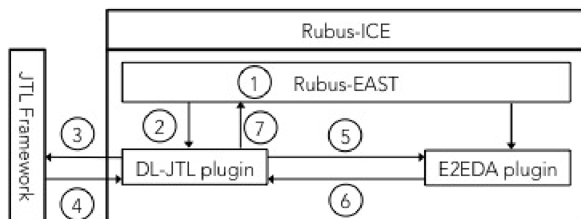

Fig. 8: Methodology instantiated within Rubus-ICE

V. Conclusion

In this work we target core challenges arising when end-to-end timing models are extracted to support end-to-end timing analysis at the design level of the EAST-ADL development methodology. Towards such goal we propose a two-phase methodology that exploits MDE, CBSE and their crossplay. Within the proposed methodology, the design-level model of the system under development is automatically transformed to all possible implementation-level models. Further, End-to-end timing analyses are performed on each generated implementation-level model; analyses results are filtered based on specified constraints and eventually the analysis results are fed back to the design-level model. Due to lack of needed information, timing model(s) can not be unambiguously extracted from a design-level model. More precisely, more than one timing model may correspond to a single design-level model, as shown in Section III. One way to deal with this issue might be to consider a priori mapping between the design-level model and one of the feasible implementation-level model. In contrast, the proposed methodology is able to generate and manage all the feasible implementation-models (transformation phase) and it is able to choose the implementation-model which better meets the timing requirements, based on the timing analysis results (timing analysis phase). Such methodology naturally supports DSE and model refinements. As a proof of concept, we instantiate the proposed methodology within the Rubus-ICE industrial tool suite. As a future investigation direction, we will, together with our industrial partners, validate, and possibly refine, such methodology upon real industrial design-level models. In this context, it is important to evaluate the performance and scalability of the proposed methodology when the number of alternatives may grow remarkably.

Acknowledgement

This work is supported by the Swedish Research Council (VR) and the Swedish Knowledge Foundation (KKS) within the projects SynthSoft and FEMMVA respectively. The authors would like to thank the industrial partners Arcticus Systems and Volvo Construction Equipment, Sweden.

References

[1] T. A. Henzinger and J. Sifakis, "The Embedded Systems Design Challenge," in *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*. Springer, 2006, pp. 1–15.

[2] I. Crnkovic and M. Larsson, *Building Reliable Component-Based Software Systems*. Norwood, MA, USA: Artech House, Inc., 2002.

[3] J. Bézivin and O. Gerbé, "Towards a precise definition of the omg/mda framework," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.

[4] "EAST-ADL Domain Model Specification, Deliverable D4.1.1, 2010," http://www.atesst.org/home/liblocal/docs/ATESST2_D4.1.1_EAST-ADL2-Specification_2010-06-02.pdf.

[5] "AUTOSAR Techincal Overview, Release 4.1, Rev. 2, Ver. 1.1.0., The AUTOSAR Consortium, Oct., 2013," http://autosar.org.

[6] "TIMMO Methodology, Version 2, Deliverable 7, Oct. 2009."

[7] "TIMMO-2-USE," http://www.timmo-2-use.org/.

[8] "TADL: Timing Augmented Description Language, Version 2, Deliverable 6, Oct. 2009," The TIMMO Consortium.

[9] Timing Augmented Description Language (TADL2) syntax, semantics, metamodel Ver. 2, Deliverable 11, Aug. 2012.

[10] K. Hänninen et.al., "The Rubus Component Model for Resource Constrained Real-Time Systems," in *3rd IEEE International Symposium on Industrial Embedded Systems, 2008*, Jun. 2008.

[11] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study," in *Computer Science and Information Systems, vol. 10, no. 1, pp 453-482, Jan. 2013*.

[12] S. Mubeen, J. Mäki-Turja and M. Sjödin, "Towards Extraction of Interoperable Timing Models from Component-Based Vehicular Distributed Embedded Systems," in *International Conference on Information Technology: New Generations (ITNG)*, Apr. 2014.

[13] A. Bucaioni, A. Cicchetti, and M. Sjödin, "Towards a metamodel for the rubus component model," in *1st International Workshop on Model-Driven Engineering for Component-Based Software Systems*, Sep. 2014. [Online]. Available: http://www.es.mdh.se/publications/3676-

[14] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Translating timing constraints during vehicular distributed embedded systems development," in *1st International Workshop on Model-Driven Engineering for Component-Based Software Systems*, Sep. 2014.

[15] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Softw.*, vol. 20, no. 5, pp. 42–45, 2003. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1231150

[16] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Jtl: A bidirectional and change propagating transformation language," in *Software Language Engineering*, 2011, vol. 6563, pp. 183–202.

[17] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming." MIT Press, 1988, pp. 1070–1080.

[18] J. Denil, A. Cicchetti, M. Biehl, P. D. Meulenaere, R. Eramo, S. Demeyer, and H. Vangheluwe, "Automatic deployment space exploration using refinement transformations," *Electronic Communications of the EASST*, vol. Recent Advances in MPM, no. 50, Jun. 2012.

[19] K. Vanherpen, J. Denil, P. De Meulenaere, and H. Vangheluwe, "Design-space exploration in model driven engineering," SOCS-TR-2014.4, McGill University, Tech. Rep., 2014.

[20] A. Hegedus, A. Horvath, I. Rath, and D. Varro, "A model-driven framework for guided design space exploration," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11, 2011.