Anticipating Implementation Level Timing Analysis for Driving Design Level Decisions in EAST-ADL

Alessio Bucaioni[1,2], Antonio Cicchetti[1], Federico Ciccozzi[1], Romina Eramo[3], Saad Mubeen[1], and Mikael Sjödin[1]

[1] Mälardalen University, Västerås, Sweden
[name.surname]@mdh.se
[2] Arcticus Systems AB, Järfälla, Sweden
[3] University of L'Aquila, L'Aquila, Italy
romina.eramo@univaq.it

Abstract. The adoption of model-driven engineering in the automotive domain resulted in the standardization of a layered architectural description language, namely EAST-ADL, which provides means for enforcing abstraction and separation of concerns, but no support for automation among its abstraction levels. This support is particularly helpful when manual transitions among levels are tedious and error-prone. This is the case of design and implementation levels. Certain fundamental analyses (e.g., timing), which have a significant impact on design decisions, give precise results only if performed on implementation level models, which are currently created manually by the developer. Dealing with complex systems, this task becomes soon overwhelming leading to the creation of a subset of models based on the developers experience; relevant implementation level models may therefore be missed. In this work, we describe means for automation between EAST-ADL design and implementation levels to anticipate end-to-end delay analysis at design level for driving design decisions.

1 Introduction

The importance of software is growing in practically all industrial sectors. In the automotive domain, software is used, e.g., for improving the safety of the vehicle, the driving experience, and the comfort of the passengers. The electronic system of a modern car can be composed of more than 70 embedded systems running up to 100 million lines of code [1]. As a consequence, development of these systems is a daunting task. Especially painful is to make late discoveries, during testing, that the software system does not deliver a service of acceptable quality w.r.t. timing errors and delays that cause suboptimal performance of important systems such as engine- or stability-control. Thus, *early analysis* of expected timing-behaviors and feasibility of architectural decisions w.r.t. timing requirements would be very welcome as support for design decisions. In this paper we propose a technique to achieve early *timing*[4] analysis.

Among the many methodologies advocating abstraction, separation of concerns, and automation as powerful instruments for dealing with complexity

[4] Although other relevant extra-functional properties and related analyses exist, the focus of this work is on timing-related properties and analysis.

of software development, Model-Driven Engineering (MDE) has progressively gained industrial attention in the past 15 years [2]. In automotive, the adoption of MDE resulted in the standardization of a layered architectural description language, namely EAST-ADL [3].

EAST-ADL proposes a top-down approach relying on four different abstraction levels, i.e., vehicle, analysis, design and implementation, and it provides abstraction and implicitly ensures separation of concerns through the different engineering phases[5]. Each abstraction level, except implementation, is equipped with a specific modeling language. At implementation level EAST-ADL proposes the adoption of existing modeling languages, e.g., AUTOSAR[6] or the Rubus Component Model (RCM) [4]. Due to its high precision timing analysis [5], we consider RCM as the reference modeling language exploited at implementation level. EAST-ADL provides mediums for achieving abstraction and separation of concerns, but it does not come with explicit support for automation among the different abstraction levels. The lack of this crucial means, imperative for a full-fledged MDE approach, leads to a scattered development process where consistency among artefacts is a burden for the developer to bear.

Due to the lack of detailed timing information (e.g., control flow ports, clocks, to mention few) [5] at design level, timing analysis cannot be performed on design models, which indeed need to be translated to implementation models equipped with needed timing details (e.g., clocks). This translation is usually done manually, driven by the developer's experience and, due to size and complexity of the task, it often considers a one-to-one mapping only. This, besides being tedious and error-prone, may lead to the loss of relevant implementation-model candidates when dealing with complex industrial systems.

In this work, we discuss a methodology which provides automation means for seamlessly linking EAST-ADL design and implementation levels to enable end-to-end delay analysis at design level[7] for supporting design decisions. The importance of exploiting implementation level analysis for taking design decisions resides in the fact that it is *more accurate* than design level analysis, which usually provides estimations and does not suffice industrial needs. The initial idea was introduced in [6], while in this work we focus on its *enhancement*, *concrete implementation* and *deployment* in the automotive context.

The rest of the paper is organized as follows. In Section 2 we present related work documented in the literature. In Section 3 we describe a running example taken from the automotive domain, and in Section 4 we apply the proposed methodology to it. In Section 5 we discuss benefits and limitations of the proposed methodology and conclude the paper in Section 6.

2 Related Work

Model-based approaches supporting timing analyses can be distinguished between those detached from design models, e.g. [7], and those deriving (part of) the necessary information from the design, like [8, 5]. In general, the latter have the advantage of avoiding *discontinuities* due to the abstraction gap between design and analysis [9], even though they have to deal with the intrinsic issue of evaluating multiple implementation choices [10, 11]. Some approaches propose manual mappings to reduce uncertainty between architectural and intermediate models, which is tedious and error-prone when dealing with hundreds of implementation alternatives. Other approaches introduce automation by specifying a

[5] In the remainder of the paper we will refer to design level models simply as *design models* and to implementation level models as *implementation models*.

[6] `http://www.autosar.org/`

[7] For *design level* we mean the EAST-ADL design level throughout the paper.

predefined one-to-one mapping between architectural and intermediate model elements, like [12] and in a broader way the refinement process prescribed by the Model-Driven Architecture standard[8]. Even though this alleviates time and error-proneness issues of manual approaches, it still relies on a predefined mapping, while in general different implementation alternatives, for the same design, should be evaluated [11].

Our solution proposes to generate a set of possible implementations, each of which entailing (possibly) different timing characteristics. Then, end-to-end delay analysis is run to evaluate them in terms of their timing characteristics and to select the best candidate(s). In this way, relevant design decisions can be anticipated before the final implementation is reached. It is worth noting that a similar mechanism could be realized, notably, by adopting other non-bijective transformation languages, architectural languages (e.g., AADL [13]), and/or other model-based timing analyses approaches (e.g., Simulink[9] or MARTE[10]). However, some preconditions should hold: i) the transformation language should fully support non-bijectivitness; ii) the architectural language shall provide adequate support for timing information at design level of abstraction; iii) the timing analyses shall keep their reliability by relying on the sole design level information (plus the alternatives generated during the derivation process).

The mechanism of implementation models generation resembles the general concept of design-space exploration (DSE) [14], and in particular rule-based DSE [15]. Our approach performs an exhaustive generation of implementation models, enriched with timing details, as derivable from the system architecture designed through EAST-ADL, and constrained by domain-specific rules. Therefore, as opposed to typical DSE, the generation is not meant to provide optimization hints at architectural level [12], rather it shows the best (timing configuration) result given a certain system architecture as input. This procedure is technically identified as quality-driven model transformations [16, 17].

3 A Running Example: the Steer-by-wire System

A steering system in a vehicle employs mechanical and hydraulic components between wheels and steering wheel. The Steer-by-wire (SBW) system, which we leverage as running example, replaces most of these components with electronic ones. We model the SBW system at the EAST-ADL design level with the help of the Rubus-ICE[11] tool suite. In the hierarchy of a design model, the leaf element is the so-called *design function prototype* (DFP). EAST-ADL implements the type-prototype mechanism, meaning that a DFP represents a specific instance of *design function type*, which defines the type. Within EAST-ADL, DFPs communicate through *function ports*, which are linked via *function connectors*.

It should be noted that one of the main goals of this example is to demonstrate the validity of the proposed methodology. Therefore, in order to better understand the transformation and corresponding selection process, we only consider the internal software architecture of the SC_ECU as depicted in Figure 1. The internal software architecture of the SC_ECU consists of six DFPs.

`Steer_Angle` is responsible for acquiring the steer angle sensor input. It passes the acquired values to `Steer_Angle_Preprocessing`. The preprocessed steer angle signal is passed to`Input_Processing`. which also receives the speed

[8] `http://www.omg.org/mda/`

[9] `http://www.mathworks.com/products/simulink/`

[10] `http://www.omg.org/spec/MARTE/`

[11] `http://www.arcticus-systems.com`

of the vehicle from `Vehicle_Speed`. `Input_Processing` passes the processed input data to `FB_Steer_Torque_Computation`, which in turn produces the feedback steering torque and passes it to `Steer_Sensation_Actuator`, which produces the signals for the steering actuator.

The WCETs specified on `Steer_Angle`, `Steer_Angle_Preporcessing`, `Input_Processing`, `Vehicle_Speed`, `FB_Steer_Torque_Computation` and `Steer_Sensation_Actuator` are 120, 200, 280, 120, 1200 and 100 μs, respectively. Since the implementation details are not available at the design level, the WCETs are estimated based on the expert's judgements. The following timing requirement is specified too:

– *"The calculated age and reaction delays shall not exceed 25 ms and 35 ms, respectively."*

Within EAST-ADL, timing requirements are specified by timing constraints [18]. Therefore, there are two end-to-end delay constraints, namely age and reaction, specified on the software architecture of the SC_ECU as shown in Figure 1. The values of the age and reaction constraints are 25 ms and 35 ms respectively.

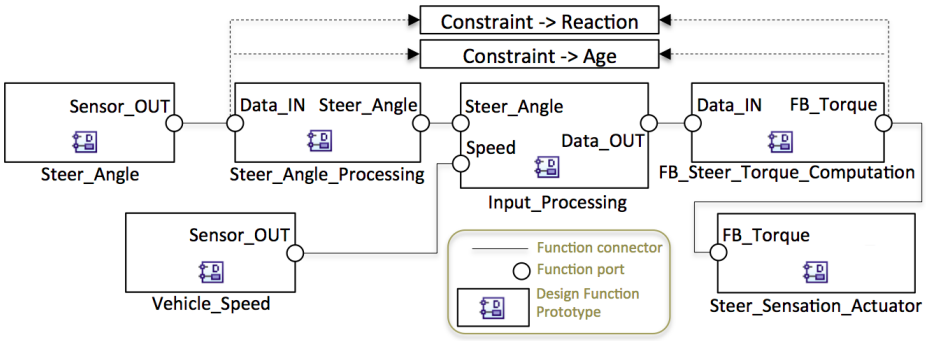

Fig. 1: Internal software architecture of SC_ECU at design level.

4 Applying the methodology

Design models do not contain the timing information (e.g., control flow) needed for running end-to-end delay analysis. Therefore, in order to leverage this analysis at design level, we propose to automatically translate design to implementation models, which contain the needed timing information. Such a translation is non-bijective, meaning that multiple implementation models can be valid translations of a given design model. To this end, the proposed methodology generates all the meaningful (from an analysis perspective) implementation models.

The approach, depicted in Figure 2, leverages the interplay of model-driven techniques and model-based analysis and it consists of four main phases, namely *transformation*, *end-to-end delay analysis*, *filtering* and *propagation*. Starting from a design model of an automotive functionality, the approach generates a set of corresponding meaningful implementation models (*transformation phase*, 1 in Figure 2) enriched with timing elements whose values are set at generation time by the developer or via configuration files. At this point, end-to-end delay analysis is run on the generated models resulting in a set of analysis results (*end-to-end delay analysis phase*, 2 in Figure 2). These results are checked against a non-empty set of timing constraints derived from the timing requirements expressed on the vehicle functionality. The result which better meets the given

timing constraints is selected (*filtering phase*, 3 in Figure 2); note that multiple results might be equally good and thereby selected. Eventually, the selected candidates are propagated back to the design level by means of annotations to the design model (*propagation phase*, 4 in Figure 2).

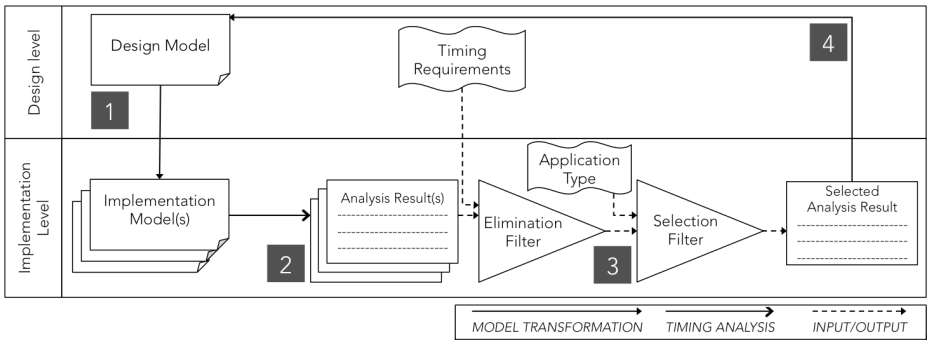

Fig. 2: Methodology supporting delay analysis at design level.

4.1 Transformation Phase

The *transformation phase* relies on a model-to-model transformation, called DL2RCM, between the EAST-ADL design level and RCM metamodels. DL2RCM is a non-bijective transformation realized within the Eclipse Modeling Framework (EMF)[12] using the Janus Transformation Language (JTL) [19].

JTL is a constraint-based bidirectional model transformation language specifically tailored to support non-bijectivity by generating all the possible solutions at once. It adopts a QVTr-like syntax and allows a declarative specification of relationships between MOF models. The language supports object pattern matching, and implicitly creates traces to record what occurred during a transformation execution. The JTL implementation relies on the Answer Set Programming (ASP) [20], which is a type of declarative programming able to address hard (primarily NP-hard) search problems and based on the model (answer set) semantics of logic programming. The ASP solver finds and generates, in a single execution, all the possible models which are consistent with the transformation rules by a deductive process.

The DL2RCM transformation consists of 28 rules mapping design elements to correspondent implementation elements. In the hierarchy of an RCM implementation model, which represents the transformation's output format, a *software circuit* (SWC) is the leaf element and encapsulates basic software functions. RCM distinguishes between data and control flow therefore a SWC has *data port* and *trigger port*. Within RCM, *Data connectors* link data ports while *Trigger connectors* link trigger ports. *Clocks* and *trigger sinks* are used to initiate and terminate the execution of a SWC, respectively.

Listing 1.1 depicts a fragment of the *DL2RCM* transformation[13], which is expressed in the textual concrete syntax of JTL and applied on models given by means of their Ecore representation in EMF. In particular, the following rules are defined:

[12] `http://www.eclipse.org/modeling/emf/`

[13] Implementation available at `http://jtl.di.univaq.it/downloads/DL2RCM.zip`

– *C2C*, which maps a function connector to both a data and trigger connectors and triggers the transformation of the connected DFPs;
– *E2C*, which maps a DFP, connected via a function connector, to a SWC;
– *E2CCS*, which maps a DFP, connected via a function connector, to a SWC equipped with a clock and a sink.

The *when* and *where* clauses specify conditions on the relation. For instance, the *where* clause on Line 17 selects the *function ports* linked by the considered *function connector* and triggers the subsequent rules.

E2C and E2CCS define a non-bijective portion of the transformation. In fact, a DFP connected via a connector may be mapped to either a SWC or a SWC equipped with a clock and a sink. This means that, from one single design model, the transformation is able to generate multiple implementation models, each of which containing a unique control flow.

```
1 transformation DL2RCM(dl:designlevel, rcm:RCM) {
2  relation C2C {
3   name, id: String;
4   checkonly domain dl con : designlevel::FunctionConnector {
5    name=name,
6    id=id
7   };
8   enforce domain rcm a : RCM::Assembly {
9    connectorData = cd:RCM::ConnectorData {
10    name=name,
11    id=id+"_d",
12    sourcePort = RCM::PortDataOut { ... },
13    targetPort = RCM::PortDataIn { ... }
14   },
15   connectorTrig = ...
16  };
17  where { (con.ends->select(end | end.functionPort.oclIsKindOf(designlevel::
        FunctionFlowPort) and end.designFunctionPrototype.isOfType.isElementary
        =true)->forAll(end | E2C(end,a) and E2CCS(end,a)  )); }
18 }
19 relation E2CCS {
20  name2, id2: String;
21  checkonly domain dl e : designlevel::FunctionConnectorInstanceReference {
22
23   designFunctionPrototype = dfp :designlevel::DesignFunctionPrototype {
24    name=name2,
25    id=id2
26   }};
27  enforce domain rcm a : RCM::Assembly {
28   clock = clk: RCM::Clock {
29    name=name2+'_clock',
30    name=id2+'_clock'
31   },
32   sink = snk: RCM::Sink {
33    name=name2+'_sink',
34    name=id2+'_sink'
35   },
36   circuit = cir :RCM::Circuit {
37      name=name2,
38      id=id2,
39      interface = int :RCM::Interface {
40       name=name2+'_interface',
41       id=id2+'_interface'
42      }}};
43  where { ... }}
44    relation E2C {
45  ...
46 }}
```

Listing 1.1: Fragment of the DL2RCM transformation in JTL.

The DL2RCM model transformation, applied to our design model in Figure 1, generates 64 implementation models [14] (one of them is depicted in Figure 3).

[14] Each SWC can be transformed either via the E2C rule or via the E2CCS rule.

However, considering the end-to-end delay analysis we want to perform, we are only interested in the combinations of those DFPs that are enclosed by the start and end points of the timing constraints. To this end, we added an OCL logic constraint (shown in Listing 1.2) to the DL2RCM transformation for reducing the set of generated implementation models. It imposes the selection of the implementation model alternatives in which *Steer_Angle*, *Vehicle_Speed* and *Steering_Sensation_Actuator* are transformed by the *E2CCS* rule.

```
Circuit.allInstances()->excluding(self.getConstrainedSWC())->select(c:Circuit
      | c.getClock().oclIsUndefined() and c.getSink().oclIsUndefined())
```

Listing 1.2: Logic constraint applied to the DL2RCM transformation.

Therefore by enforcing the bijectivity on the *Steer_Angle*, *Vehicle_Speed* and *Steering_Sensation_Actuator*, the DL2RCM transformation generates 8 implementation models[15].

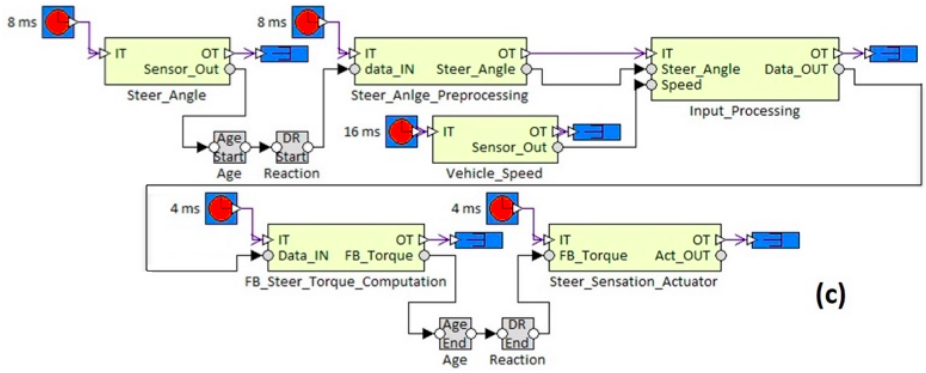

Fig. 3: Generated implementation model example.

4.2 End-to-end Delay Analysis Phase

In this phase, we predict the timing behavior of each generated implementation model by performing the end-to-end delay analysis [21, 5]. We are interested in the calculations of two different delays, namely *age* and *reaction* [5]. Age delay is important in control applications where the interest lies in the freshness of received data. Reaction delay is used to determine the first reaction time for a given stimulus. Our focus is on the Controller Area Network (CAN) which is a event-triggered serial communication bus protocol. We do not use global time stamps (that require tracking of global chronological time) to predict the timing behavior. Instead we use response-time analysis and end-to-end delay analysis. We refer the reader to [21, 5] for the details about the calculations of age and reaction delays.

Once the analysis has been performed on each generated implementation model, the analysis results, which include calculated age and reaction delays for each individual implementation model as shown in Table 1, are forwarded to the filtering phase.

For calculating age and reaction delays, the methodology employs the timing analysis engines implemented in the Rubus-ICE.

[15] All the combinations of the *Steer_Angle_Preprocessing*, *Input_Processing* and *FB_Steering_Torque_Computation* are generated by not enforcing bijectivity.

		Delay Analysis (μs)				**Delay Analysis** (μs)	
		Age Delay	*Reaction Delay*			*Age Delay*	*Reaction Delay*
Model	*(a)*	26020	30020	**Model**	*(e)*	26020	30020
	(b)	26020	42020		*(f)*	26020	42020
	(c)	18020	22010		*(g)*	18020	18020
	(d)	2020	10020		*(h)*	18020	18020

Table 1: Delay Analysis Result for the generated implementation models.

4.3 Filtering and Propagation Phases

The filtering phase consists of two cascaded filters: the *elimination filter* and the *selection filter*. The timing analysis results are provided as input to the elimination filter together with the non-empty set of timing constraints. In our example, the elimination filter compares the analysis results of each implementation model with the specified age and reaction constraints of 25 and 35 ms respectively. The implementation models identified as (a), (b), (e) and (f) in Table 1 violate one or both timing constraints; hence, they are discarded. The remaining models, which satisfy the specified timing constraints (i.e., **(c)**, (d), (g) and (h)), are forwarded to the selection filter.

The selection filter selects the best implementation model based on the requirement concerning the type of application, also received as input. To this end, an application i) contains only single-rate chains, or ii) contains multi-rate chains. In our example, the system shall be developed using multi-rate chains. This means that the implementation models that contain single-rate chains between start and end points of the specified timing constraints are negligible. Therefore, the models identified in Table 1 as **(c)**, depicted in Figure 3, and (g) are selected[16]. Finally, the models and their analysis results are propagated back to the design model (as annotations done by text-to-model transformations).

5 Discussion

Running and leveraging implementation level analysis at higher abstraction levels (e.g., design) brings multiple advantages. First of all, it can help the designer in taking architectural decisions based on much more precise feedback than common design level analysis, which, being based on estimated or guessed properties, are usually just conceived as complementary to implementation level analysis in industrial settings. Moreover, it allows the developer to only focus on design activities exploiting implementation level analysis results without having to investigate nor manually edit implementation models, which are automatically produced and transparent to the developer.

We employ JTL to generate multiple implementation models from one design model by providing different combinations of implementation elements, derived from the design model, and timing elements, added by the transformation. Clearly, the generation of all possible combinations, besides being unnecessary in most scenarios, becomes soon unbearable from a scalability perspective when dealing with complex systems of industrial size. For this reason, we exploit JTL's capability of entailing ASP logic constraints for narrowing the generation space.

[16] The selection filter selects the implementation model with shorter age and reaction delays. In our case two models have same analysis results, thus they are both selected.

We provide a set of default constraints to prune solutions that are evidently meaningless for our analysis. This means that we can enable support for the generation of different classes of models by providing different default constraints. Nonetheless, default constraints do not prevent the generation of dimly meaningless solutions nor high transformation time in case of very complex design models. While the first issue can be solved through analysis and filtering mechanisms, the latter demands additional user-defined constraining based on the specific modeled functionality.

It is interesting to note that the methodology may propagate more than one generated implementation model, along with its timing analysis results, to the design model. This happens only when those results are equally good. In this case, the designer is given the possibility to select among them.

By considering the general development scenario, through our methodology it is possible to disclose the opportunity of shortening time-to-market and leverage expensive resources (e.g., architects, timing experts) more efficiently. More concretely, the simple software system illustrated in this work contains more than fifty components, seventeen in the SC_ECU and ten in each of the four WC_ECUs. This means that starting from such an architecture a designer willing to manually define a proper implementation model would face a space of 2^{57} possible alternatives. It becomes evident that having an automated mechanism that is able to derive those alternatives and select the best one(s) brings a gain in terms of time, costs and risks in the construction of the implementation.

6 Conclusion

The approach proposed in this paper tackles the problem of identifying a suitable implementation choice, in terms of timing characteristics, starting from the software architecture. In general this issue requires the consideration of a number of alternatives that grows exponentially with the number of software components in the architecture. We proposed to solve this by adopting a *quality-driven model transformation* approach and defining a precise mapping between EAST-ADL design and implementation models (defined in terms of the Rubus Component Model). Since in general the mapping of design to implementation models equipped with timing elements is non-bijective, we leveraged the properties of a constraint-based transformation language, JTL, to automatically derive all the meaningful implementation alternatives. Subsequently, generated implementation models are classified in terms of timing results enabling the selection of the best implementation model candidate(s) derivable from the input design model.

The experiment we conducted in collaboration with industrial partners in automotive showed promising results w.r.t. time gains and reduction of possible errors in the creation of a suitable implementation model. Despite the generation and selection processes are transparent to the developer, issues about scalability remain open. In particular, the size of the problem could reach a point such that the generation of implementation alternatives would be intractable. In this respect, a main future investigation direction encompasses the study of smarter generation rules. Another line of research will be devoted to the study of combining the optimisation of multiple system (especially extra-functional) properties.

Acknowledgement

This work is supported by ARTEMIS, the Swedish Research Council (VR), the Swedish Foundation for Strategic Research (SSF), and the Knowledge Foundation (KKS) with the projects CRYSTAL, SynthSoft, PRESS and SMARTCore. The authors would like to thank the industrial partners Arcticus Systems AB and Volvo AB, Sweden.

References

1. Robert N. Charette. This Car Runs on Code. *Spectrum, IEEE*, 46(2), 2009.
2. D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39:25–31, 2006.
3. 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.
4. K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K.-L. Lundback. The rubus component model for resource constrained real-time systems. In *Procs of SIES*, pages 177–183, June 2008.
5. 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. *Computer Science and Information Systems*, 10(1), 2013.
6. A. Bucaioni, S. Mubeen, A. Cicchetti, and M. Sjödin. Exploring timing model extractions at east-adl design-level using model transformations. In *Procs of ITNG*, April 2015.
7. M. Gonzalez Harbour, J.J. Gutierrez Garcia, J.C. Palencia Gutierrez, and J.M. Drake Moyano. Mast: Modeling and analysis suite for real time applications. In *Procs of ECRTS*, pages 125–134, 2001.
8. S. Anssi, S. Tucci-Piergiovanni, C. Mraidha, A. Albinet, F. Terrier, and S. Gérard. Completing east-adl2 with marte for enabling scheduling analysis for automotive applications. In *Procs of ERTS*, 2010.
9. B. Selic and L. Motus. Using models in real-time software design. *Control Systems, IEEE*, 23(3):31–42, June 2003.
10. B. Schatz, F. Holzl, and T. Lundkvist. Design-space exploration through constraint-based model-transformation. In *Procs of ECBS*, pages 173–182, March 2010.
11. J. Denil, A. Cicchetti, M. Biehl, P. De Meulenaere, R. Eramo, S. Demeyer, and H. Vangheluwe. Automatic deployment space exploration using refinement transformations. *EASST*, Recent Advances in MPM, 2012.
12. M. Walker, M.-O. Reiser, S. Tucci-Piergiovanni, Y. Papadopoulos, H. Lnn, C. Mraidha, D. Parker, D. Chen, and D. Servat. Automatic optimisation of system architectures using east-adl. *Journal of Systems and Software*, 86(10):2467 – 2487, 2013.
13. Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report SEI Technical Note CMU/SEI-2006-TN-011, 2006.
14. M Gries. Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38(2):131–183, December 2004.
15. A. Hegedus, A. Horvath, I. Rath, and D. Varro. A model-driven framework for guided design space exploration. In *Procs of ASE*, 2011.
16. J. Merilinna. A Tool for Quality-Driven Architecture Model Transformation, 2005.
17. M.L. Drago, C. Ghezzi, and R. Mirandola. Towards quality driven exploration of model transformation spaces. In *Procs of MoDELS*, pages 2–16. 2011.
18. Timing Augmented Description Language (TADL2) syntax, semantics, metamodel Ver. 2, Deliverable 11, Aug. 2012.
19. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Jtl: a bidirectional and change propagating transformation language. In *Procs of SLE*, pages 183–202. 2011.
20. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Procs. of the ICLP 1988*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.
21. N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *Procs of CRTS*, 2008.