Coupling of Timing Properties for Embedded Realtime Systems Using A Hybrid Tool Integration Approach

Arne Noyer, Padma Iyenghar, Elke Pulvermueller
Institute for Software Engineering
University of Osnabrueck, Germany
{anoyer, piyengha, elke.pulvermueller}@uos.de

Joachim Engelhardt, Gert Bikker
Institute for Distributed Systems
Ostfalia University of Applied Sciences, Germany
{jo.engelhardt, g.bikker}@ostfalia.de

***Abstract*—The increasing complexity of realtime embedded software systems necessitates them to support a variety of functions. Therefore, a practice is to split the overall system into different sub-systems and use various (modeling) tools for describing different parts of the system. On the other hand, though guaranteeing realtime properties forms an integral part of realtime embedded software development, it is often overlooked during system specification. Thereby, much effort is needed to solve timing problems, when they occur after implementing a system. Therefore, it is desirable to specify and validate timing properties during early development stages. As different parts of an overall system can be developed using various tools, elements which influence the timing behavior can be located in different sources. In order to analyze the overall timing behavior with a timing validation tool, all timing properties of the overall system are needed. Therefore, this paper presents an approach to couple timing properties from different sources. The approach allows to create links between elements across different sources and to synchronize timing properties. Thereby, timing properties can be followed, kept consistent and combined to be validated.**

Keywords*—*Timing Properties; Tool Integration; Model Transformations; Unified Modeling Language (UML); MARTE

I. Introduction

Model Driven Development (MDD) can be considered as an ongoing paradigm shift to address the steadily increasing complexities involved in developing software systems, in particular also applicable to realtime and embedded software development. Various modeling languages and tools have been established for realizing different parts of a system using the MDD. An example of a widely adopted modeling domain is the Unified Modeling Language (UML) [1].

An embedded software system employing one modeling domain may comprise of several sub-systems developed using multiple models. The sub-systems may communicate among themselves, to fulfill an overall functionality. As an example, let us consider the functionality for power window in an automotive. Let us say that on one Electronic Control Unit (ECU), there are two sub-systems, one each controlling the motors for the windows and for clamping protection respectively. These two sub-systems must communicate among themselves to achieve the overall functionality of the power window. Such sub-systems with related timing properties may be developed across different teams/partners; thereby requiring an appropriate coupling of the timing properties before performing a timing validation of the overall software system.

978-1-5090-1314-2/16$31.00

Models can also be used for specification of/capturing non-functional requirements/quality attributes (e.g. timing requirements). The compliance of timing requirements is often safety critical and violating them can lead to a system failure and have catastrophic results. When such failures are only detected late when the combined code of all sub-systems is running on a target device, it is often hard to analyze the cause of the timing errors. Therefore, it is desirable to specify timing behavior for different parts directly in the models while development.

In the models, as they describe parts of the same overall system, there can be intersections. Thus, it is important to keep timing properties consistent which are present in different models. In addition, there can be specialized timing models for specifying timing properties of a system. Here, it is desirable that a connection can be established between the element in one model and a corresponding timing property in the other model. Specialized timing validation tools such as the SymTA/S [2] employ sophisticated techniques [3] for predicting the timing behavior based on its own timing model. In order to analyze the overall timing behavior of a system, it is desirable to couple/synchronize the timing properties from different sources and keep them consistent among each other and with a validation model. However, such an overall approach for linking/merging and synchronization of overlapping properties spread across different sources is still missing. Addressing this gap, this paper presents a tool integration approach for coupling (i.e., linking, merging and synchronization) of intersecting timing properties across various sources (in this paper: sub-systems in several UML models) and then to validate the timing properties of the overall software system.

The remaining of this paper is organized as follows. Section II presents the overall approach. Afterwards, in section III, an experimental evaluation is made and related work is outlined in section IV. Section V concludes the paper.

II. Proposed Approach

In this section, the proposed approach for coupling timing properties of embedded realtime systems, which are spread across different sources (i.e., sub-systems), is outlined. The approach takes into consideration that the timing properties

could be spread across different sources (e.g. models in MDD) during development. After the timing properties of a whole system are coupled from different sources, an overall timing analysis can be made with a validation tool like SymTA/S [2].

Please note that in the context of this paper, we consider the *coupling* aspect with the following semantics: the timing properties from various sources (e.g. domains/models/sub-systems) can be related to each other. Further, the timing properties which are not only loosely related but are semantically the same, are additionally synchronized with each other. In addition, all the timing information from every sub-system (model/domain) is merged together in an overall timing model in order to use it for an analysis of the whole system.

A. Overview

The workflow proposed in this paper for coupling the timing properties from various sources (e.g. various sub-systems) is shown in Fig. 1. Here, two UML models are used to describe parts of a software system. Further, elements in the UML models are annotated with timing properties. An example pertaining to coupling of timing properties among two different sub-systems, developed with UML, is used to describe the proposed approach. The timing properties in the UML models can be specified using extensions for UML such as the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile [4].

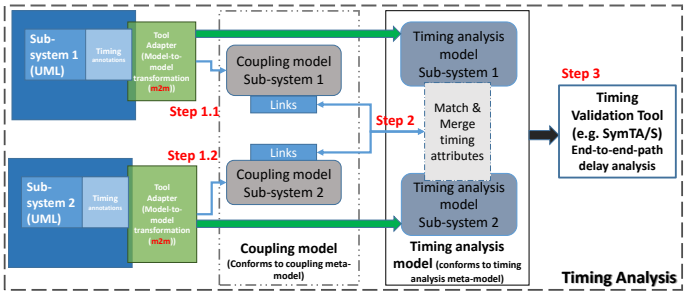

Fig. 1. Workflow for merging and validating timing properties

In step 1.1 and 1.2 in Fig. 1, model transformations are executed that create timing and coupling models. While the coupling models contain information about original UML elements with timing properties, the timing models contain the resulting timing elements. In addition, links between timing model elements and coupling model elements are created for identifying the original UML elements. In step 2, the timing models are merged in order to allow a timing validation of the overall system in step 3.

The basis for the approach are the two metamodels, the *coupling* metamodel and the *timing* metamodel, which are used for exchanging and connecting the timing properties. In the existing approaches either all the data from various sources involved are stored in a single metamodel. Otherwise, only the links are stored in a single metamodel leaving out the data (i.e., the timing attributes). These two cases are avoided in the proposed approach. The proposed timing metamodel is used to store the timing properties only in a common form and the coupling model enables linking of the related timing properties. In this context, the proposed approach is termed as a *hybrid tool integration approach* in this paper.

B. Timing metamodel and Coupling metamodel

Fig. 2 illustrates an excerpt of the timing metamodel, which is used to store the related timing properties in a common model for an overall system. There are *Packages* for structuring elements like *Cores*, *Tasks*, *ECUs* and *Runnables* (i.e., operations). Further, the element *ExecutionPath* is used to specify the timing properties for the steps involved and the order of the steps for realizing a functionality. An example of the execution path is the steps involved in implementing a brake application functionality of a car (from pressing the brake pedal to stopping the car). The meta-model is similar to the model in SymTA/S [2], as it is a state of the art tool for validating timing behavior. However, the purpose of the timing metamodel is to have a uniform model, for which model to model transformations can be implemented in order to exchange timing properties between tools.

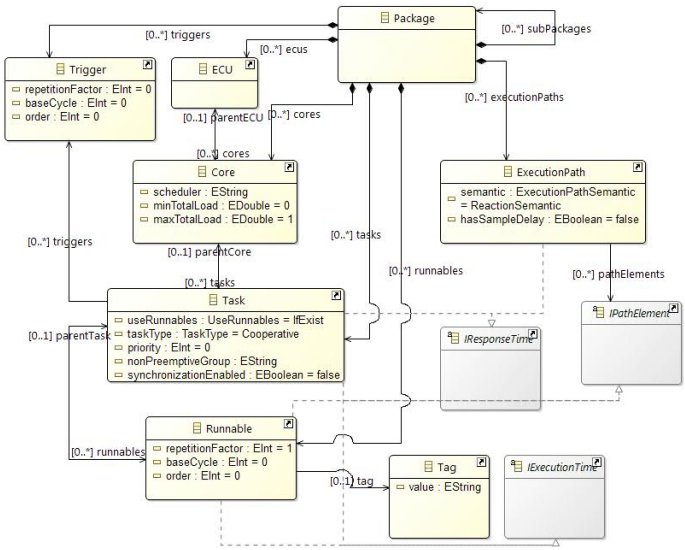

Fig. 2. Excerpt of the timing metamodel

The main elements of the coupling metamodel are depicted in Fig. 3. An *ExternalSource* element is created for every source of timing properties like a UML model. Thereby, the field *connection* is used to store a string value to identify the represented source element. In case of a UML model, the value of this field is a path to the model file.

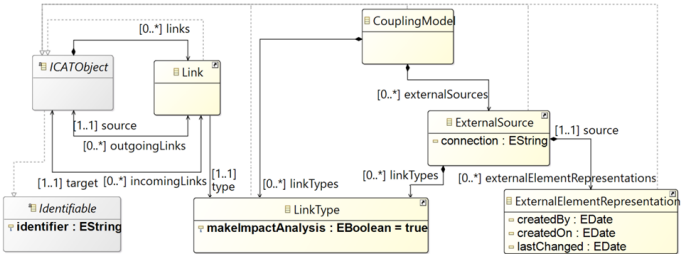

Fig. 3. Main elements of the coupling metamodel

Similar to the *ExternalSource* element representing a source of timing properties, the element *ExternalElementRepresentation* represents the elements within the source, for instance a UML class in a UML model. Again, there is a string field to store which element it represents from the source. To

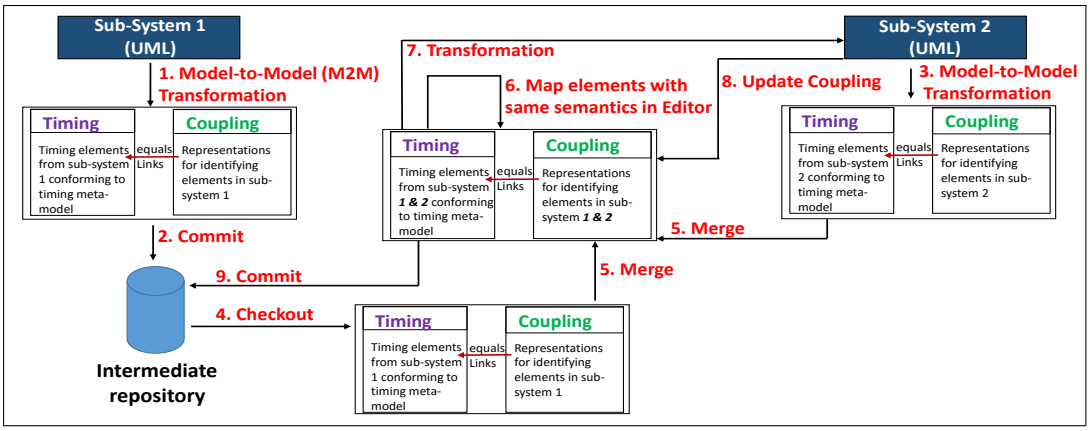

Fig. 4. Steps for merging timing properties with an intermediate repository

represent an UML element, this field contains an identifier to uniquely identify the element in the source model. For instance, the UML tool Rhapsody [5] has unique identifiers for every UML element, which would be stored in this field.

By managing representations of elements in a common form, the coupling model can be used to create *Links* between elements of various sources. Further, the links can have a *LinkType* to differentiate between different kinds of links. Users can create custom LinkTypes, but by default, the LinkTypes *refines*, *validates*, *equals* and *represents* are created.

A link always references a *source* and a *target* element which is of type *ICATObject* (Coupling And Traceability Object). Not only is the *ExternalElementRepresentation* inherited from *ICATObject*, but also every element from the timing metamodel (Fig. 2). Therefore, after a M2M transformation is executed in the workflow in Fig. 1, a link can be created between an *ExternalElementRepresentation* representing an element from a model and an element in the common timing model. Then, the link will have a type which indicates that the elements are equal. Please note that the elements in the common timing model can also be linked with each other.

C. Merging and synchronizing timing properties

Ensuring consistency of timing properties across various sources (e.g. sub-systems from one modeling domain) is a challenging and a non-trivial task. Certain timing properties could be exclusive in one model and there can be an intersection of them between models, which raises the need for synchronizing them. Therefore, the common, *intermediate timing model* is persisted so that it can be used for continuously merging and synchronizing timing properties between various sources. In addition, this makes it possible to access timing properties in a common form, even if a source of the timing properties is not available. For instance, this can be the case if a tool which is a source of timing properties is running on a server that can not be accessed at a required instant. The steps for merging timing properties in a common repository from models containing two sub-systems, in the proposed approach, are illustrated in Fig. 4 and described in the following:

Step 1: When the intermediate repository with timing information is empty, the first step is to make a M2M transformation from *one* source of timing properties (e.g. sub-system 1 in Fig. 4). This would result in an instance of the timing metamodel and the coupling metamodel. The timing model contains the timing properties and elements from the source in a common form. The coupling model contains representations for the corresponding elements in the source. Further, links are created between the *ExternalElementRepresentations* and the timing model elements to create a connection between the elements from the source model and the timing model.

Step 2: The newly created timing and coupling models (e.g. corresponding to sub-system 1, 2 in Fig. 4) are committed to a repository (or persisted in another form). This is the basis for further integration of timing properties from other sources.

Step 3: Let us consider that once the above steps are completed, we have a scenario for integrating of timing properties from another source and synchronization of the second source. For example, let us say that the timing and coupling model are already available for sub-system 1 in Fig. 4. Now, there arises a case wherein the timing properties from another source, e.g. sub-system 2 must be integrated with the timing properties of sub-system 1 (which already exists in the repository from step 1). To enable this, a M2M transformation is performed for creating an instance of the timing and coupling model for the sub-system 2. From this step to the last step, the repository is locked for being changed by any other accesses than in these steps. The steps 3-9 must be repeated when timing attributes from any further sources (sub-systems from same/different domains) need to be coupled using the proposed approach.

Step 4: The fourth step is to checkout the persisted timing and coupling models in order to be able to merge them with those from the sub-system 2 in the next step.

Step 5: Merging of the two timing models (one each from sub-system 1 and sub-system 2) and two coupling models is performed in step 5. The merged models then contain representations and timing elements/properties both of sub-system 1 and sub-system 2. As there can be an intersection between the two timing models, which means that there were elements in modeling domain 1 and 2 that have the same meaning, there can be redundant elements. For instance, the same CPU core could be created in two different UML models, while different tasks are allocated to it.

Step 6: The user can select several timing elements, which are resulting from modeling domain 1 and 2, in an editor and

specify that these having an *equals* relationship. Afterwards, equal timing elements are merged to a single element. For instance, this can be one CPU Core which is specified in two different sub-systems (with various tasks being mapped to it). When a conflict arises in the properties among the two CPU elements, when they are merged to a single element, it is up to the user to choose one among them. For future work, it is also planned to investigate automatic merging strategies. However, after elements from different models are mapped to be equals, the information is used for any further synchronization and the equals relationships do not have to be defined each time.

Step 7: Since the mapping and merging (steps 5-6) may have resulted in changes to the timing model, these changes have to be reflected in the sub-system 2 (as mentioned in step 3). This is necessary to be consistent with the timing model again. Therefore, as a seventh step, a M2M transformation is performed from the merged models to sub-system 2. While doing this, the *ExternalElementRepresentations* are used to identify the elements in sub-system 2, which are *equals* to elements in the timing model.

Step 8: If the elements are created or deleted in the sub-system 2 during the previous step, this will affect the *ExternelElementRepresentations* in the coupling model. Therefore, after updating the elements in the sub-system 2, the intermediate coupling model has to be updated in this step.

Step 9: Finally, the coupling model and timing model are consistent to the content in the second sub-sytem and the updated coupling model and timing model can be persisted in the repository. If the first sub-system (or any other source) needs to commit the timing information to the repository from now on, the steps 3-9 are executed for it, too. This ensures that the timing information in sub-system 2 is synchronized and the overall timing model updated.

The aforementioned steps 1-9 in Fig. 4 constitute the step 2 in Fig. 1. In the next step, the overall timing model can be subject to a timing validation (i.e., step 3 in Fig. 1).

III. Experimental Evaluation

A real-life embedded software system example based on an electric screwdriver is used as for the experimental evaluation of the proposed approach. The software system comprises of two main sub-systems created using two UML models as seen in Fig. 5-(a), (b) in an UML tool [5]. The UML

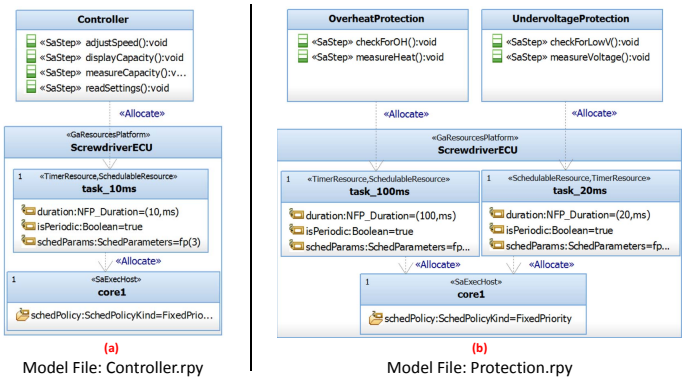

Fig. 5. Two UML models for the screwdriver in IBM Rational Rhapsody

model in Fig. 5-(a) implements a class *Controller* for adjusting the speed of the screwdriver. The second model in Fig. 5-(b) implements the classes for realizing mechanisms such as overheat protection and undervoltage protection. In both the models, a class *ScrewdriverECU* is used for representing the microcontroller in which the overall software is running. In addittion, this class contains a CPU core, *core1*. Therefore, the elements *ScrewDriverECU* and *core1* are redundant in both UML models. Further, the classes *Controller*, *Undervoltage-Protection* and *OverheatProtection* are mapped to tasks and the tasks are mapped to run on the same CPU core of the microcontroller. The tasks are executed periodically and may have different execution durations.

The timing attributes for the UML models in Fig. 5 are specified using the MARTE profile. The table in Fig. 6 provides a list with some of the stereotypes which are used from MARTE, for annotating the screwdriver example with timing constraints. For timing analysis, elements such as

Stereotype	Applied on	Tagged Values	Description
SaAnalysisContext	Packages	isSched, workload, platform	Entry point for analysis models
GaResourcePlatform	Classes	resources	The platform which is analyzed
SaExecHost	Classes, Objects	mainScheduler, schedPolicy, utilization, isSched	CPU Cores
Scheduler	Classes, Objects	schedPolicy	Scheduling mechanism
SchedulableResource	Classes, Objects	schedParams, host	Tasks
TimerResource	Classes, Objects	duration, isPeriodic	Applied to periodic tasks
SaStep	Operations	deadline, concurRes, priority, execTime	Step in an execution path
Allocate	Dependencies		Allocation of elements to others
GaWorkloadBehavior	Activites, Interactions	behavior, demand	The behavior for the workload
GaScenario	Interactions	cause	Different execution paths
SaEndToEndFlow	Activity Partition, Interactions	end2EndT, end2EndD, isSched	Complete end to end flow, contains reference to GaScenario
GaWorkloadEvent	AcceptEvent-Actions, Messages	pattern, effect	Defining a pattern for execution paths

Fig. 6. Stereotypes used from MARTE for the screwdriver example

tasks (*SchedulableResource*), CPU cores (*SaExecHosts*) and a scheduling mechanism are defined using these stereotypes. The elements are placed in a UML package, in which the stereotype *SaAnalysisContext* is applied to indicate that they are relevant for scheduling analyses. In addition, execution paths are created using UML sequence diagrams in order to define the execution order of operations. Applying the stereotype *SaStep* on the operations allows for defining a minimum and maximum execution time of the operations and their priorities.

Applying the proposed approach, a M2M transformation is performed for each UML model in the screwdriver example. The transformations are implemented using the Atlas Transformation Language (ATL) [6]. The above step results in a timing and coupling model, one each for the two UML models (in Fig. 5 -(a), (b)). Excerpts of the resulting timing and coupling models are shown in Fig. 7.

In the timing models, there are elements for each UML element which are relevant to timing analysis conforming to the timing metamodel. For instance, for the operations with the stereotype *SaStep*, elements of type *Runnable* are created. For the classes, which contain the operations/runnables, *Software Components* are placed in the timing models and the runnables are mapped to them corresponding to the UML model.

In addition to the timing models, coupling models (Coupling and Traceability (CAT) Model in Fig. 7) are created. Thereby, elements of type *ExternalSource* are used to identify

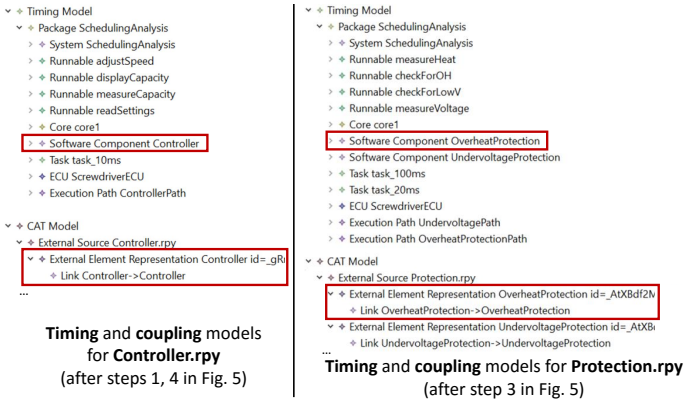

Fig. 7. Timing and coupling models for the two UML models

the UML models. For identification purpose, these elements store the path to the UML models. In order to identify contained elements, which were transformed to elements in the timing model, *External Element Representations* are created. These *External Element Representations* are related to the representing UML elements by storing the identifier of the UML element with which it is identified in the UML tool [5]. The connection between the *External Element Representation* and the timing elements are established by *Links*. For instance, for the class Controller in the Rhapsody model Controller.rpy there is an *External Element Representation* that has the id of the Rhapsody Element and contains a link which references the *Software Component Controller* inside the timing model. The *External Element Representation* and the *Link* for the *Controller* and *Overheatprotection* are highlighted in Fig. 7.

After the timing and coupling models were created, merging has to be made (step 5 in Fig. 4). As a result, the timing and coupling models of the two UML models are combined and then a single timing model and a single coupling model exists, which is shown on the left side in Fig. 8. Inside the coupling model, there are *External Source* elements for both UML models with all contained *External Element Representations*. Similarly, the timing model contains all elements which were previously in two separate timing models. The combination of the models resulted in redundant elements like *ScrewdriverECU* and *core1* in the timing model, as they were created from elements in two different UML models. Inside the different UML models, they had the same meaning but weren't the same elements and had different identifiers and therefore, it is hard to generically identify the resulting timing elements as being the same. Therefore, we leave it up to the user for now to select elements which mean the same in the model editor (automatic merging strategies are future work). After the user selected elements to be the same, they are combined to one and the *Link*s in the *External Element Representations* are updated, too. If there is a conflict with certain attribute values, the user further has to select which values to keep. All other steps like executing the model transformations are performed automatically. On the right side in Fig. 8 the models are shown after mapping identical elements in the left models. Therefore, in the right models there is for instance only a single *core1* and *ScrewdriverECU* and no redundant elements. If elements do not have the same meaning but are related to each other, the user can create *Links* between them and assign a link type which indicates the characteristic of the relationship. By creating a link, it can be i.e. differentiated if one value is estimated or measured by assigning a corresponding link type.

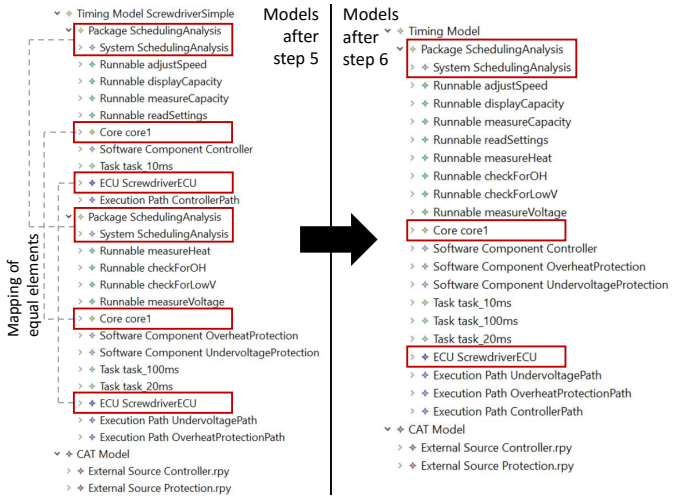

Fig. 8. Combining the timing and coupling models of the two UML models

There are still some steps left for merging which are illustrated in Fig. 4. In the example above, the models on the left side in Fig. 7 are not only resulting from one UML model, but represent the up-to-date models in the repository, before merging the timing information from the second UML model. After merging, the updated timing and coupling models have to be committed to the repository. However, since the merging may have changed timing elements which are in the second UML model, a M2M transformation from the merged models to the second UML model is performed (step 7 in Fig. 4). Of course, elements from the first UML model could have been changed as well, but in the steps 4-9, it is about synchronizing the second UML model with the repository. Changes in elements, which affect the first UML model, are updated in it when synchronizing it the next time.

While executing the model transformations in step 7, the coupling model is used to identify elements in the UML model which are related to elements in the timing model. If UML elements are created or deleted during the transformations, the coupling model is updated accordingly (step 8 in fig. 4). Finally, the overall, merged coupling and timing models are committed to the repository and other tools can access them and/or also merge and synchronize their timing properties similarly with the steps 4-9 from Fig. 4. In order to perform a timing analysis for the overall timing model with the validation tool SymTA/S, model transformations from/to it were implemented to exchange information with the repository like the UML model described above.

IV. Related Work

In this section related work pertaining to model-based specification and coupling of timing attributes in embedded software are discussed.

In the UML domain, the MARTE profile [4] [7], is a standard for specifying the timing behavior for real-time embedded software systems. Apart from the UML domain, several

modeling/domain-specific languages and a number of generic approaches have emerged that include timing behavior. Some examples are the TIMMO (TIMing MOdel) and TIMMO-2-USE projects, which were developed as a timing extension for EASTADL2 [8]. The automotive standard AUTOSAR 4 [9] has its own timing model, which was influenced much by the results of the TIMMO project. In addition, AMALTHEA [10] provides common a meta-model for multi-core software and hardware modeling. Further, it enables different tools of a customized tool chain to gain access to the overall, unified model. However, it does not provide possibilities for coupling timing information from different sources. In [11], a model-based workflow from specification until validation of timing requirements is introduced.

There are several generic approaches for data integration of tools. A common approach for data integration is to persist *all* data from different tools in a single repository. By this method, all the participating tools in data integration can access any data. An example for this is GeneralStore [12]. In addition to persisting all the data in a single repository, there are approaches for synchronizing models of tools using M2M transformations as in [13] [14]. The Open Services for Lifecycle Collaboration (OSLC) [15] is an emerging tool integration approach which allows the tools to access each others' data by using web-technologies. Further, links can be created between elements across different tools and links can have different link types. Another approach which allows to create links across different models is described in [16]. In this, a (conceptual) layer with virtual models is created above the ordinary models like UML. In the virtual models, links can be created across models. However, this results in needing new toolings, which allow to work on the virtual models instead of the ordinary models. The proposed approach is a combination of concepts, which store data in a common form (for synchronization) and allow for creating links across models. Compared to other concepts, it has several advantages that are discussed in the conclusion.

V. Conclusion

The presented approach enables to couple related timing properties of an overall system from different tools / sources. This enables a meaningful validation of the timing behavior of the overall software system. In order to couple the timing properties and for synchronization, two models, a *timing* model and a *coupling* model, are used and persisted in a central repository. When *synchronizing* and *merging* timing properties with the central repository, model transformations are executed to create a *coupling* and *timing* model and then merge them with those in the central repository. By always using intermediate models and not creating model transformations between any tool combinations, an advantage of the approach is that it is generic and has a high degree of scalability.

The approach is termed a hybrid tool integration approach, since unlike approaches with a common repository for all data from all tools, not all information is synchronized with the repository. Only the relevant information for timing analysis is synchronized. There is no metamodel which consists of all elements from every compatible tool, but only a metamodel for timing information. Hence, this is another advantage regarding the scalability. Further, the approach is termed hybrid because the coupling metamodel not only allows for identifying elements for synchronization, but also allows to create links between elements from different tools. However, it is not always needed that every element with timing information is available in every tool. Often, it is sufficient to relate elements to each other using links. Elements are only synchronized when there are *equals* relationship between a timing element in the intermediate model and several elements. Other link types than *equals* can be used to indicate different characteristics of values like *estimated* and *measured*. Therefore, the approach has the advantage that both is possible: creating links across tools and synchronization via a central repository.

Storing timing properties in a central repository results in further advantages. Thereby, timing properties are made available, even if data from a tool cannot be accessed (e.g. is offline). In addition, the intermediate timing and coupling models can be exchanged between partners, if they develop different parts of a system. However, after all timing properties of an overall system are coupled, a timing analysis can be made by using a validation tool like SymTA/S. The proposed approach is validated by using an example embedded software system, which is developed using two UML models.

Some future directions: automatic merging strategies; to use OSLC for creating links and data access; to validate the approach with more tools/sources for timing properties.

References

[1] "Unified Modeling Language (UML)," http://www.uml.org/.

[2] Symtavision GmbH, "SymTA/S & TraceAnalyzer," https://www.symtavision.com/products/symtas-traceanalyzer/.

[3] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis-the symta/s approach." IEEE Proceedings-Computers and Digital Techniques 152.2, 2005.

[4] Object Management Group, "MARTE Specification 1.1," 2011, http://www.omg.org/omgmarte/Specification.htm.

[5] IBM, "Rational Rhapsody Family Website," accessed 04/2016. [Online]. Available: http://www-03.ibm.com/software/products/en/ratirhapfami

[6] Atlas Transformation Language (ATL), https://eclipse.org/atl/.

[7] B. Selic and S. Gerard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE - Developing Cyber-Physical Systems*. Morgan Kaufmann Publishers, 2014.

[8] A. Consortium, "East-ADL2 specification," ITEA, Tech. Rep., 2008.

[9] AUTOSAR Standard, http://www.autosar.org/, 2016.

[10] AMALTHEA Project, https://itea3.org/project/amalthea.html, 2016.

[11] A. Noyer, P. Iyenghar, E. Pulvermueller, J. Engelhardt, F. Pramme, and G. Bikker, "A model-based workflow from specification until validation of timing requirements in embedded software systems," in *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*. IEEE, 2015, pp. 1–4.

[12] C. Reichmann, M. Kiihl, P. Graf, and K. D. Mller-Glaser, "Generalstore - a case-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems," in *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*. IEEE, 2004, pp. 225–232.

[13] C. Hein, T. Ritter, and M. Wagner, "Model-driven tool integration with modelbus," in *Future Trends of Model-Driven Development*, 2009.

[14] K. Ehrig, G. Taentzer, and D. Varró, "Tool integration by model transformations based on the eclipse modeling framework," *EASST Newsletter*, vol. 12, pp. 1861–0668, 2006.

[15] Open Services for Lifecycle Collaboration (OSLC) Community, "OSLC," http://open-services.net/.

[16] C. Guychard, S. Guerin, A. Koudri, A. Beugnard, and F. Dagnat, "Conceptual interoperability through models federation," in *Semantic Information Federation Community Workshop*, 2013.