Specifying Intra-Component Dependencies for Synthesizing Component Behaviors

Stefan Dziwok^{1,2}, Sebastian Goschin³, and Steffen Becker¹

- Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Zukunftsmeile 1, 33102 Paderborn, Germany, stefan.dziwok@upb.de
 - ² Fraunhofer IPT, Project Group Mechatronic Systems Design, Software Engineering, Zukunftsmeile 1, 33102 Paderborn, Germany
 - ³ Step2e Innovation GmbH, Rosstraenke 4, 94032 Passau, Germany

Abstract. Cyber-physical systems, e.g., cars, interact with their physical environment, underlie real-time constraints, and exchange messages with each other. An engineer can define their software using a component-based architecture. An approach to manage the complexity of this task is to separate concerns by specifying the behavior of each component's port independently and, afterwards, synthesizing the component behavior based on the port's behaviors and their dependencies. Though, such a synthesis requires to specify the intra-component dependencies formally. However, for several dependencies that are commonly used, no formal language exists. In this paper, we present a language that enables the specification of all commonly used dependencies in the domain of cyber-physical systems. Moreover, we define the requirements for an intra-component dependency language, provide an extended synthesis process, and introduce the dependency kinds the language shall support.

Keywords: Component-Based Software Systems, Model-Driven Engineering, Model Dependencies, Cyber-Physical Systems, MECHATRONICUML

1 Introduction

Cyber-physical or intelligent technical systems [7] are systems that interact with their physical environment. Their software development is especially difficult as these systems have to obey real-time constraints, are safety-critical, and coordinate with other systems by exchanging various messages. A cyber-physical system (CPS) might utilize a component-based software architecture, where components interact via ports. The interaction is either message-based or signal-based (e.g., to access sensors, actors, and feedback controllers). The behavior specification of a component is expressed via state machines. In addition, to ensure a safe and useful coordination between the components, each interaction between message ports has to adhere to communication protocols at application-level that define which message at which time span must be sent or received.

One particular example of CPSs are autonomously driving cars. In the scenario given in Fig. 1, the white car is faster than the gray car and, thus, wants to



Fig. 1. Overtaking Scenario: The white car wants to overtake the gray car.

overtake it. Each car can measure its velocity and white can additionally measure its distance to gray. To reduce the risk of a crash, the involved systems shall coordinate the overtaking by adhering to two communication protocols. Both cars must adhere to the communication protocol Overtaking which formally defines that white needs permission from gray to start the overtaking and that gray does not accelerate while white is overtaking. Moreover, white and a track information provider (TIP) must adhere to the communication protocol TrackInfo which formally defines that the TIP informs white whether the track is safe (e.g., it is not safe if obstacles are on the street). To further increase the safety, white shall fulfill additional requirements: (d1) white may only request to overtake, if it reaches the safety distance to gray and the TIP has confirmed that the track is safe for more than 5s. (d2) The planned overtaking speed that white sends to grey depends on the current velocity of white. (d3) Before informing gray that the overtaking has finished, white must restore the safety distance to gray.

A software engineer that shall develop the component behavior for the white car's software can split this task into two steps to enable a separation of concerns and, therefore, to manage the task's complexity. In step 1, he specifies the externally visible component behavior. It consists of the behavior of the component's single ports to access physical information (here: the distance and the velocity) and to exchange messages according to the two communication protocols. The result is a set of independent port behavior specifications. Step 2 serves to additionally fulfill the intra-component requirements (here: d1-d3). To achieve this, the software engineer has to enhance the independently defined behavior models of the component. Thus, these requirements are intra-component dependencies. As errors in the component behavior are safety-critical, it is of vital importance that the software engineer considers all intra-component dependencies and implements them correctly such that the component behavior adheres to both, the single ports behaviors as well as the ports communication requirements. However, our experience shows that executing the second step manually is – despite this separation of concerns – still complex and error-prone.

A semi-automatic approach for step 2 is provided by Eckardt and Henkler [6]. They provide domain-specific languages (DSLs) a software engineer can use to formally describe the intra-component dependencies between the independent behaviors. Based on the independent behaviors and the formal dependencies, the component behavior can be synthesized automatically. The approach is integrated into the software engineering method MECHATRONICUML [4,8].

In this paper, we extend the approach of Eckardt and Henkler [6] as their formal dependency languages are not able to express all intra-component dependencies that are commonly used in the domain of CPSs. For example, their languages cannot express the mentioned dependencies d1-3. One reason is that their languages are not able to access the physical information (distance and velocity). As the related work also does not provide DSLs for the unsupported dependencies, we define a new DSL. In particular, the contribution of our paper is as follows: (i) we explicitly define the requirements for specifying intra-component dependencies that shall serve as an input for the synthesis, (ii) we extend the synthesis process defined by Eckardt and Henkler, (iii) we provide 20 kinds of intra-component dependencies, and (iv) we introduce a formal intra-component dependency language for MECHATRONICUML. We provide our implementations and the models of our running example online⁴.

The paper is structured as follows. Section 2 introduces MECHATRONICUML and presents the models of the running example. Then, in Sect. 3, we define the requirements for an intra-component dependency language and discuss to what extent Eckardt and Henkler's approach fulfills them. We explain the adapted synthesis process in Sect. 5. Afterwards, we present the identified kinds of intra-component dependencies in Sect. 6 and introduce our new DSL in Sect. 7. We discuss related work in Sect. 8 and conclude the paper in Sect. 9.

2 MechatronicUML

MECHATRONICUML [4,8] is a method that is designed for the software development of CPSs. It provides a component-based modeling language and a development process. Thus, it is able to specify the models of the overtaking scenario.

MECHATRONICUML distinguishes between continuous and discrete components. Continuous components represent, among others, sensors, actors and timecontinuous feedback-controllers. In contrary, the behavior of discrete components is specified via extended states machines called Real-Time Statecharts (RTSCs). A RTSC may contain data variables and continuous clocks (known from timed automata [3]) which are used to define real-time constraints. Clocks can be reset to zero and be compared with an expression. An example for a discrete component is WhiteSw (cf. Fig. 2). Discrete components may contain discrete ports to exchange messages with other discrete components and hybrid ports to send and receive signals from continuous components. A RTSC of a discrete component can specify to send and receive messages via its discrete ports and may read or write values from hybrid ports. In our example, WhiteSw contains the discrete ports Overtaker and Receiver and the hybrid ports Distance and Velocity. In particular, Distance is a hybrid in-port that periodically reads an incoming signal measuring the distance between the cars while Velocity is a hybrid out-port that writes an outgoing signal to set the car's target velocity.

For ensuring a correct message communication, each discrete port must adhere to application-level communication protocols called Real-Time Coordina-

⁴ https://trac.cs.upb.de/mechatronicuml/wiki/ModComp2014

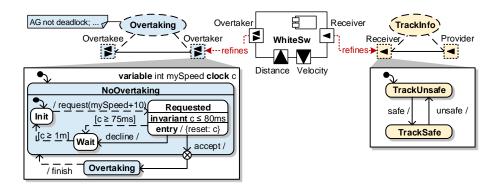


Fig. 2. Models of the Overtaking Scenario Specified in MECHATRONICUML

tion Protocols (RTCPs). In particular, a RTCP consists of two communicating partners, called roles, whereby each role represents a discrete port. As depicted in Fig. 2, in our example, the discrete port Overtaker must adhere to and may refine the same-named role of RTCP Overtaking, which coordinates the overtaking. The discrete port Receiver must adhere to and may refine the same-named role of RTCP TrackInfo, which coordinates the information whether the track is safe. RTCPs may be formally verified via timed model checking [3] concerning formal properties, e.g., that a deadlock may never occurs. Due to the separation of components from RTCPs, MECHATRONICUML enables a scalable timed model checking of the software under development for large and complex systems [8].

The behavior of a RTCP depends on the behavior of each roles which we separately specify using RTSCs. The lower part of Fig. 2 shows the RTSCs for the roles Overtaker and Receiver. The informal behavior description of the role Overtaker is as follows: Initially, the role is in state NoOvertaking.Init and may at any time send an overtaking request to role Overtakee including its planned overtaking speed as a message parameter which is the current speed plus 10 km/h. Within 75 ms, it expects an accept or decline message. If no message or a decline message is received, the role Overtaker has to wait until 1 minute in total is over before it may send a new request. However, if the message accept is received, it switches to state Overtaking and may now start the overtaking process. After the overtaking, it sends the message finish and switches back to NoOvertaking.Init. In contrast, the behavior of the role Receiver is quite simple: Initially, the role Receiver assumes that the track is unsafe. As soon as the track is safe, the role Receiver will be informed by the role Provider about this change. Analogously, the role Provider will inform the role Receiver when the track becomes unsafe.

3 Existing Dependency Languages for MechatronicUML

Eckardt and Henkler [6] provide for MECHATRONICUML two formal languages to describe intra-component dependencies: state composition rules (SCR) and event composition automata (ECA).

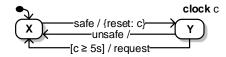


Fig. 3. Event Composition Automata for Synchronizing the Message Exchange

SCRs are able to synchronize the concurrent executed behavior of discrete ports based on their states and clocks. In particular, SCRs can forbid that a set of states may be active together at any point in time or at least for a specific time interval (by defining constraints that reference existing clocks of a port). An exemplary SCR is as follows: $\neg((A, true) \land (B, c < 30))$. It defines that the states A and B may not be active together when the clock c is less than 30.

ECAs are expressed using timed automata [3]. They are able to synchronize the behavior of two discrete ports based on sequences of messages the ports exchange with other components. Moreover, timing restrictions on the message sequences may be defined using auxiliary clocks. In particular, the automaton can enforce the order and the delay of messages that the ports can send or receive. We show an example in Fig. 3. The depicted automata constraints the sending of message request by enforcing that message safe shall be received at least 5 seconds before and that message unsafe is not received in between.

4 Requirements for the Dependency Language

Before we adapt the existing dependency languages, we explicitly define requirements and discuss the current fulfillment by Eckardt and Henkler [6].

Based on our experience, the dependency language shall (r1) be formal, (r2) be able to reference existing modeling elements of the design language that are used in the independent behaviors, (r3) include analyses to identify errors and conflicts within the specification, (r4) cover the most commonly used dependencies, (r5) be acceptable by all stakeholders, (r6) only enable useful dependencies, and (r7) have a good tool support with a high usability.

The requirements r1 and r2 are essential to enable an automated synthesis, while r3 shall improve the correctness of the dependencies and thus avoid senseless syntheses. Concerning r4, in our opinion, the goal should not be to support all dependencies that are theoretically possible. This would lead to a language that is complex to use and may result in incorrectly formalized dependencies. Moreover, it is hard (maybe even impossible) to prove that all possibilities are covered. Having a less complex language should benefit r5 because - in our opinion - software engineers define the formal dependencies but requirements and test engineers need to be able to understand them as they have to check if the informal requirements are covered. Via r6, we want to prevent that the engineer can define dependencies that the synthesis does not support or that he can define senseless expressions (e.g., tautologies). Concerning r7, good tool support reduces the development time and avoids mistakes by the user. Examples for usability enhancements are live syntax checks and auto suggestions.

The approach of Eckardt and Henkler [6] only fulfills the requirement r1. The other requirements are partially or not fulfilled. The reason for this is that their focus was to prove that their general synthesis approach is applicable rather than concentrating on its usability or its completeness. In particular, they only partially fulfill r2 as their languages cannot reference all syntax elements of RTSCs, e.g., hierarchical states, concurrent state machines, variables, actions, and entry/do/exit state events. Moreover, hybrid ports cannot be referenced, too. Requirement r3 is not fulfilled as they only focus to identify conflicts after the synthesis. Concerning requirement r4, their two dependency languages already support a lot of dependencies. However, the languages do not support commonly used dependencies, e.g., dependencies concerning data variables. Without an evaluation, we can not answer in general if requirement r5 is fulfilled. However, in our opinion, it is not obvious which dependencies they describe. This leads to the conclusion that r5 can still be improved. Requirement r6 is only partially fulfilled as they only enable dependencies that the synthesis supports but their language allows superfluous dependencies (e.g., tautologies). Concerning requirement r7, good tool support exists for ECA in form of a graphical editor. However, SCR dependencies are defined as a plain string only and, therefore, do not provide usability enhancements like the ones mentioned above.

5 Adaptions to the Synthesis Process

As most of the requirements from Sect. 4 were not in Eckhardt's and Henkler's focus, adaptation to their syntheses process [6] are necessary. Figure 4 shows the new process for semi-automatically defining the component behavior. In particular, we added the steps 1b and 2c and the analyses for the steps 2a and 2c. In the following, we briefly describe the new process.

Based on (informal) requirements and the component structure, in the Steps 1a and 1b, software engineers concurrently specify all independent behaviors that form the external component behavior, i.e., RTCPs for discrete ports and hybrid ports. For each RTCP, a developer has to apply timed model checking to ensure their correctness concerning the specified properties (e.g., no deadlock occurs). Then, in Step 2a, a software engineer formalizes the dependencies using a DSL that fulfills the requirements from Sect. 4. Dependencies that cannot be formalized using the DSL are postponed to Step 2c. While specifying, analyses identify errors and conflicts (e.g., contradictions and tautologies) to prevent superfluous executions of the synthesis. Afterwards, in Step 2b, the developer executes the automatic synthesis and the succeeding refinement checks [11] which verifies if the discrete ports still adhere to their RTCP. In Step 2c, a software engineer may adapt the synthesized component behavior to integrate the dependencies that were not formalizable in Step 2a. In that case, another refinement check is necessary. To prevent that the manual changes are lost after a new synthesis run, we propose to automatically log all manual changes and to enable a semi-automatic reapplication. The final result of this process is the complete component behavior that respects all requirements.

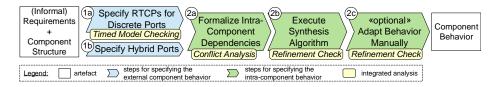


Fig. 4. Adapted Process for Synthesizing the Component Behavior

As MECHATRONICUML already fully enables the Steps 1a-b (including the timed model checking) as well as the refinement check [11], we only have to adapt the concepts of Steps 2a and 2b. In this paper, we only focus on improving Step 2a without the conflict analysis. Thus, in the following two sections, we present the most commonly used dependency kinds of MECHATRONICUML components and propose to exchange MECHATRONICUML's formal dependency languages.

6 Kinds of Intra-Component Dependencies

By analyzing the modeling language and existing example models as well as interviewing Mechatronicum users, we identified 20 commonly used dependency kinds and list them in a catalog [9, Appx. B]. We classify them concerning the involved modeling elements (data, time, transition, state, message) and their direction (uni- or bidirectional). We describe each dependency kind via an uniform description format that defines six attributes: (1) the name of the kind, (2) an informal description, (3), the involved modeling elements, (4) the direction, (5) an informal example, and (6) the formalization of this example. In the following, we briefly introduce a selection of ten kinds due to space restrictions.

Synchronization The RTSCs of two discrete ports shall synchronize their behavior by enforcing that their transitions can only fire together.

Required State Constraint A transition of a discrete port shall be constrained by the status (inactive / active) of a state of another discrete port, e.g., a transition may only fire as long as another state is active.

Data Pull A discrete port shall have read access to local data variables of another discrete port.

Data Push A discrete port needs write access to an existing local data variable of another discrete port.

Data Merge Two or more data variables are in fact the same data variable and shall be merged into one. If one of them is a hybrid port, all data variables are merged into that port.

Clock Merge Two or more local clocks of different discrete ports are in fact the same clock and shall be merged into one.

Forbidden State Combination A particular set of states distributed over several discrete ports shall not be active at the same time.

Allowed State Combination States of different discrete ports may only be active at the same time if they are listed in one common set.

Allowed Message Sequence A certain message sequence shall be allowed, but subsets of these message sequences are not allowed.

Forbidden Message Sequence A sequence of messages shall be excluded from the combined state space of the independent behaviors. However, subsets of these message sequences are allowed.

In a small survey [9, Appx. C], we verified the completeness of our catalog. The results were that no commonly used kinds were left out and most of the identified kinds are commonly used. However, the survey showed that two dependency kinds, e.g., *Allowed State Combination*, are not as common as expected. If these results are confirmed in the future, we will remove them from the catalog.

7 Dependency Languages for MechatronicUML

The dependency languages of Eckardt and Henkler [6] (partially) support 12 dependency kinds. However, among others, the kinds Synchronization, Data Push, Data Pull, Data Merge, Clock Merge, and Forbidden Message Sequence are not supported. One reason for this is that the languages do not consider data variables. Thus, we either have to extend the languages, select existing languages, or define new languages.

Instead of ECAs, we propose to use Modal Sequence Diagrams (MSD) [10] to describe message-based dependencies as they do not only support the dependencies that an ECA may specify (e.g., *Allowed Message Sequence*) but also further dependency kinds like *Forbidden Message Sequence*.

Moreover, we replace SCRs by a new DSL named MechatronicUML Intra-Component Dependency Language (MIDL). MIDL shall support all dependency kinds of SCR and all dependency kinds that MSDs do not support. We provide a detailed definition of MIDL in [9, Sect. 4.4]. Below, we list three formal dependencies for component WhiteSw specified using MIDL that relate to the informal dependencies d1-d3 given in Sect. 1. The dependencies d1 and d3 are of kind Transition Firing Constraint while dependency d2 is of kind Data Merge. The formalized dependency d1 (line 1) states that the port Overtaker may only fire the transition from NoOvertaking.Init to Requested (and send the message request) if port Receiver is in state TrackSafe for more than 5s and if the hybrid port Distance has a value below 20. The formalized dependency d2 (line 2) states that the local variable mySpeed shall be merged into the hybrid port variable Velocity. The formalized dependency d3 (line 3) states that the port Overtaker may only fire the transition from NoOvertaking to Overtaking (and send the message finish) if the hybrid port Distance has a value higher than 20.

```
1 if [Receiver.TrackSafe is active longer than 5s] and [
          Distance < 20] {enable transition Overtaker.NoOvertaking.
          Init-->Requested};
2 merge variable Overtaker.mySpeed into Velocity;
3 if [Distance > 20] {enable transition Overtaker.Overtaking-->
          NoOvertaking};
```

We realize our language by defining a formal representation using Xtext's LL(*) attribute grammar (requirement r1). We modified the generated meta

model by referencing elements of the MECHATRONICUML meta model (r2) and by adding OCL constraints that prevent erroneous and superfluous dependencies (r3,r7). Furthermore, we provide an editor for our DSL which has usability features like syntax highlighting, live syntax check, and auto completion (r7).

In a small case study about trains that may enter various track sections [9, Appx. A], we were able to formalize 29 intra-component dependencies in total from three discrete component where each component consist of three to five ports. This is a first indicator that our language fulfills r4.

8 Related Work

In the related field of controller synthesis for discrete and timed systems, formal languages are required as an input. Via scenarios, the software engineer has to define possible interactions between the controller (the software under development) and the environment, e.g., using timed game automaton [1] or using modal sequence diagrams [10]. However, the scenarios are not independent. Thus, in contrary to us, a developer does not need to specify dependencies explicitly. However, these related approaches have other negative side effects, e.g., in contrary to our approach that requires independent behaviors, they cannot enable a scalable model checking of the system which is mandatory in CPSs.

Other synthesis approaches require temporal logic expression as input, e.g., Letier et al. [12] require the Fluent Linear Temporal Logic (FLTL) to specify event-based dependencies and Attie et al. [2] propose to specify inter-task dependencies using the temporal logic CTL. Uchitel et al. [14] even support scenarios as well as FLTL as inputs for the synthesis. However, none of these logics is able to express all intra-component dependency kind that we identified for the domain of CPSs. One reason for this is that the partial behavior models of these approaches do not support all properties of a CPS (e.g., real-time, data variables, accessing physical information).

Donatelli [5] proposes to directly insert formal dependencies within the automata. This is contrary to us, as we specify our dependencies in an additional model to separate concerns. Moreover, Donatelli only enables the specification of event-based dependencies. Thus, only a small subset of our required dependency kinds is covered.

9 Conclusion and Future Work

In this paper, we defined the requirements for specifying intra-component dependencies to enable the synthesis of discrete component behaviors for CPSs. We improve an existing synthesis process and identified the most commonly used intra-component dependencies for discrete components of MECHATRONICUML. Moreover, we propose the usage of MSDs as well as a new defined DSL called MIDL to formally describe these dependencies.

Using our proposed DSL, software engineers are able to formally define intracomponent dependencies. In contrast to informally described dependencies, this should avoid misunderstandings between the engineers. Moreover, MSDs and MIDL can serve as an input for a component behavior synthesis that only requires rare manual changes afterwards. Component models like Sofa 2 [13], which define the external component behavior in separate parts and have no component behavior synthesis yet, can apply our concepts to define their own DSL for specifying intra-component dependencies. We think that a lot of the dependency kinds should still be valid and that their DSL could be similar to MIDL.

In the future, we will complete the realization of our synthesis method. First, we will further evaluate MIDL. Then, we will develop a detection of dependencies conflicts. Afterwards, we will adapt the synthesis algorithm and perform a thorough evaluation of the complete method.

Acknowledgement We thank Tobias Eckardt, Christian Heinzemann, and Hendrik Kassner for their feedback on earlier versions of this paper.

References

- Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) Hybrid Systems II, pp. 1–20. LNCS 999, Springer Berlin Heidelberg (1995)
- 2. Attie, P.C., Singh, M.P.: Specifying and enforcing intertask dependencies. In: In Proceedings of the 19th VLDB Conference. pp. 134–145 (1993)
- 3. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
- 4. Becker, S., et al.: The MechatronicUML design method process and language for platform-independent modeling. Tech. Rep. tr-ri-14-337, Heinz Nixdorf Institute, University of Paderborn (Mar 2014), version 0.4
- Donatelli, S.: Dependent automata for the modelling of dependencies. In: Setola, R., Geretshuber, S. (eds.) Critical Information Infrastructure Security, pp. 311–318. LNCS 5508, Springer Berlin Heidelberg (2009)
- Eckardt, T., Henkler, S.: Component behavior synthesis for critical systems. In: Giese, H. (ed.) ISARCS 2010. pp. 52–71. LNCS 6150, Springer (June 2010)
- 7. Gausemeier, J., Rammig, F.J., Schäfer, W. (eds.): Design Methodology for Intelligent Technical Systems. Lecture Notes in Mechanical Engineering, Springer (2014)
- 8. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. pp. 38–47. ESEC/FSE'03 (Sep 2003)
- 9. Goschin, S.: Synthesis of Extended Hierarchical Real-Time Behavior. Master's thesis, University of Paderborn (Feb 2014), http://tinyurl.com/MA-Gos14
- Greenyer, J.: Scenario-based Design of Mechatronic Systems. Ph.D. thesis, University of Paderborn (Oct 2011)
- 11. Heinzemann, C., Brenner, C., Dziwok, S., Schäfer, W.: Automata-based refinement checking for real-time systems. Computer Science Research and Development (2014), published Online June 2014
- 12. Letier, E., Kramer, J., Magee, J., Uchitel, S.: Deriving event-based transition systems from goal-oriented requirements models. Automated Software Engineering 15(2), 175–206 (Jun 2008)
- 13. Malohlava, M., Hnetynka, P., Bures, T.: Sofa 2 component framework and its ecosystem. Electr. Notes Theor. Comput. Sci. 295, 101–106 (2013)
- 14. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of partial behavior models from properties and scenarios. IEEE Transactions on SE 35(3), 384–406 (2009)