Sequence Diagram Test Case Specification and Virtual Integration Analysis using Timed-Arc Petri Nets

Sven Sieverding
OFFIS
Oldenburg, Germany
sieverding@offis.de

Christian Ellen
OFFIS
Oldenburg, Germany
ellen@offis.de

Peter Battram
OFFIS
Oldenburg, Germany
battram@offis.de

In this paper, we formally define Test Case Sequence Diagrams (TCSD) as an easy-to-use means to specify test cases for components including timing constraints. These test cases are modeled using the UML2 syntax and can be specified by standard UML-modeling-tools. In a component-based design an early identification of errors can be achieved by a virtual integration of components before the actual system is build. We define such a procedure which integrates the individual test cases of the components according to the interconnections of a given architecture and checks if all specified communication sequences are consistent. Therefore, we formally define the transformation of TCSD into timed-arc Petri nets and a process for the combination of these nets. The applicability of our approach is demonstrated on an avionic use case from the ARP4761 standard.

1 Introduction

Testing is an important activity in modern embedded systems development processes, *e.g.,* ISO 26262 [8], SAE ARP4761 [1]. It comprises different level of granularity. In case of a component-based design [5] the unit-tests have to validate that each individual component fulfills its requirements. The integration-tests deal with the problems that arise through the combination of multiple components and have to ensure their correct interaction. On the highest level, the complete system has to be validated.

In this paper, we focus on two testing aspects. First, an easy-to-use specification method for unit-tests using UML2 syntax [12]. Second, we define an early virtual integration analysis on the basis of these test cases. For the test case specification, we introduce the concept of test case sequence diagrams (TCSD) as an extension of UML2 sequence diagrams. This extension allows test engineers to annotate timing constraints for messages in the test case specification. Out of these test cases we are then able to generate a formal analysis model to perform the virtual integration analysis.

The main idea of this approach is to interpret the successfully executed unit-test cases as contract specification for the component. The pair of input and expected result defines the assumption and the promise on the connected input ports and output ports respectively. On the basis of these specifications, we are analysing if test cases for different components of the same system contradict each other regarding timing behavior or the ordering of messages.

Our main contribution is the formalisation of sequence diagram-based test cases and the virtual integration analysis of the test cases. This approach is demonstrated on a well-known aerospace example from the ARP4761, the braking system control unit (BSCU). The components of the BSCU are implemented using Matlab/Simulink and sequence diagram-based test cases and the system architecture of the BSCU are modeled using IBM Rational Rhapsody. These test cases are translated into timed-arc Petri nets (TAPN) [7], which are modeled and analyzed using the tool TAPAAL [4]. For the virtual integration, we developed a prototype which is able to export the necessary information from Rhapsody. It also translates the exported TCSD into timed-arc Petri nets, which can then be analyzed by TAPAAL.

B. Buhnova, L. Happe, J. Kofroň:
Formal Engineering Approaches to Software Components
and Architectures 2013 (FESCA'13)
EPTCS 108, 2013, pp. 17–31, doi:10.4204/EPTCS.108.2

The structure of this paper is as follows: The formal definition of sequence diagrams, test case sequence diagrams, and Petri nets is described in section 2. Section 3 presents the virtual integration process, including the translation mechanism of TCSD to TAPN. The demonstration of our analysis is evaluated in section 4 and in section 5 a conclusion as well as an outlook for our future work is given.

1.1 Related work

There is a huge amount of publications available dealing with UML [12] models for test case specification. For example Sokenou [14] identified the need to include sequence diagrams and state diagrams, which are usually created in the early stages of the development, into his test sequence generation method. Also Linzhang [10] described the UML models as a natural source for test case generation, since this semi-formal modelling language is commonly used.

The formalization of sequence diagrams has been analyzed in [9, 11, 6] as well as their transformation into other specifications. For example Bowles [2] formalized sequence diagrams and translated sequence diagrams into coloured Petri nets. In this paper, we use his formalism for sequence diagrams but translate them into timed-arc Petri nets (TAPN), because of the annotated timing information. We chose Petri nets over timed automata [15] because of the simplicity to compose different Petri nets in parallel [16].

The background of this work is based-on prior work [13] in which the idea of sequence diagram-based test case specification is introduced as well as the concept for consistency analysis. This work elaborates on these ideas and creates a formal model for the test case sequence diagrams. We also define the virtual integration analysis on the formal model of TCSDs.

2 Formal Definitions

This section introduces the formal models for our virtual integration, namely sequence diagrams, timed-arc Petri nets, and test case sequence diagrams.

The basic models for UML sequence diagrams and timed-arc Petri nets are both based-on existing formal models, whereas test case sequence diagrams are a new extension to sequence diagrams specifically designed to suit the specification needs of timed behavior of a system under test.

2.1 UML Sequence Diagrams

The semantics of sequence diagrams, we use in this paper, are based-on the informal specification in the UML 2.0 superstructure [12] and the formal semantics defined in Bowles and Meedeniya [2]. In addition, we define a slightly adapted version of these semantics, named *test case sequence diagram* (TCSD), to deal with diagrams specifically designed for test cases. Figure 1 shows an example of a TCSD and introduces the basic idea of the formalization, which is based-on the different types of events on the instance lines.

Definition 1 (Sequence Diagram (based-on [2]))
A sequence diagram (SD) is a tuple $D = (d, I, E, <, \Sigma_{msg}, M, F, X, Exp)$ *where:*

- $d \in \Sigma_{name}$ *is the name of the diagram and* Σ_{name} *the set of all diagram names;*
- I *is a finite set of object instances (lifelines);*
- $E = \bigcup_{i \in I} E_i$ *is a set of events for lifeline i,* s.t. $\forall i, j \in I : E_i \cap E_j = \emptyset$*;*
- $<$ *is a set of partial orders which defines for instance line* $i \in I$ *a set:* $<_i \subseteq E_i \times E_i$*;*

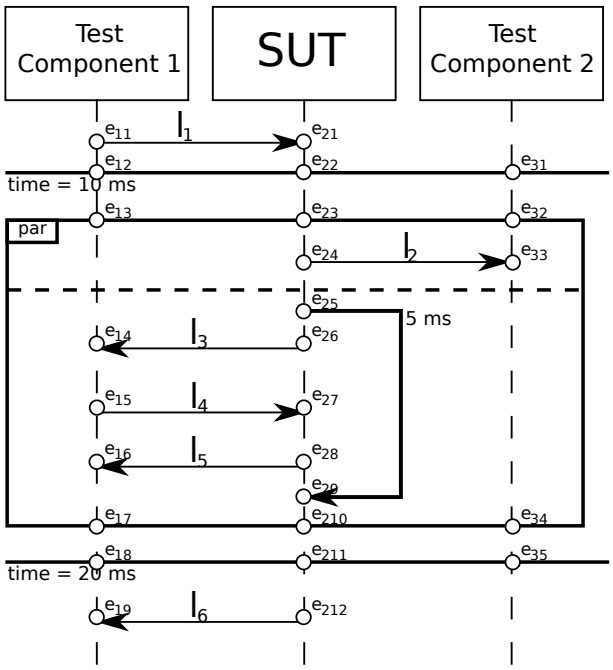


Figure 1: The figure shows examples of the structural elements of test case sequence diagrams and their different types of events.
For example: The box labeled with SUT is the instance line of the system under test, which interacts with the test components 1 & 2; the arrow labeled with l_1 is a message sent by event e_{11} and received by e_{21}; the horizontal line with events e_{12}, e_{22}, and e_{31} is a partition line which models a timing constraint of $10ms$; the box labeled with `par` is a fragment in which the events of the two operands, seperated by the dashed line, are executed in parallel; and the arrow between event e_{25} and e_{29} is a timeout operator, which constraints the time between the occurrences of these events to $5ms$.

- Σ_{msg} *is a finite set of message labels* l*;*
- M *is a set of messages* $M \subseteq E \times \Sigma_{msg} \times E$, s.t. *for every* $m_1, m_2 \in M$ *with* $m_1 = (e_{11}, l_1, e_{12})$ *and* $m_2 = (e_{21}, l_2, e_{22})$*:* $m_1 \neq m_2 \implies e_{11} \neq e_{12} \neq e_{21} \neq e_{22}$*;*
- F *is a set of interaction fragments for which the functions* op, ev, sub *are defined as:*
  - $op : F \to \Omega \times \mathbb{N}$ *associates an operator* $\Omega \in \{$`strict, par, opt, alt, loop`$\}$ *and the number of operands to a fragment;*
  - $ev : F \times \mathbb{N} \to 2^E$ *associates a set of events to a pair* (id, n) *of a fragment* $id \in F$ *and an operand index number* n*;*
  - $sub : F \times \mathbb{N} \to 2^F$ *associates a set of nested fragments to a parent fragment and an operand index number;*
- $X = \{X_i\}_{i \in I}$ *a set of local variables indexed by object instances* $i \in I$*;*
- Exp *is a set of expressions, where each expression is associated as a guard to a message or a fragment using the function* $guard : M \cup F \to Exp$

A sequence diagram as defined in Definition 1 has a name d, which is usually used to identify a diagram in a UML modeling tool and a set of object instances I. Each instance $i \in I$ describes the behavior of one object in the diagram, which is defined by the events E_i on the lifeline. All events within the diagram are partially ordered by a relation $<_i$. Only a partial ordering is possible, because the events in operands of fragment blocks like `par` cannot be ordered. An event of a sequence diagram can either be part of a message (send event/receive event) or mark the borders of an interaction fragment (enter event/exit event). The interactions of a sequence diagram are defined by its transitions (messages) and the different kind of fragments. Each message m is a tuple $m = (e_1, l_1, e_2)$ where $e_1 \in E_i$ is a send event, l_1 is a message label of the labeling alphabet Σ_{msg}, and $e_2 \in E_j$ is a receive event. Both events must be different ($e_1 \neq e_2$). In addition, the events must be ordered ($e_1 < e_2$) if they are part of the same instance line ($i = j$).

Fragments are regions within a sequence diagram with a specific semantic defined by the operator Ω of the fragment. A fragment spans over a subset of all instance lines of the diagram and every instance part of the fragment has dedicated `enter` and `exit` events which signal the fragments boundaries with respect to this instance. The operators `strict, par, alt,opt,` and `loop` are relevant for our virtual integration scenario. The behavior of `strict` is the default behavior and requires that the events on the lifeline must occur in the specified order (according to the $<_i$-relation). A `par` fragment has at least two operands. Starting with the `enter` event, the event sequences of all operands are executed in parallel. The `exit` event of the `par` fragment is reached when all operands have reached this event following the $<_i$-relation. In contrast to the parallel execution of the `par` fragment, the operands of the `alt` fragment (at least two) represent exclusive alternative event sequences. The optional behavior of the `opt` fragment can be considered as a special case of an `alt` fragment, with one operand and an implicit empty sequence as second alternative. The `loop` fragment has exactly one operand. Its sequence of events is repeated as often as stated in the expression of its guard $Exp(f_{loop})$.

For each fragment three functions *op*, *ev*, and *sub* are defined:

op : The *op* function assigns to each fragment an operator and the number of operands within the fragment. Each operator of a fragment requires a specific (minimum) number of operands. The `strict`, `loop`, and `opt` for example require exactly one operand, while the `par` and `alt` operators require at least two operands.

ev : The *ev* function defines which elements are part of an operand. Therefore, it maps a tuple (f,n) of a fragment f and an operand number n to a set of events.

sub : In sequence diagrams fragments may be nested. The function *sub* describes this hierarchy by mapping a tuple (f,n) to the set of the directly nested fragments within the nth fragment and therefore establishes a parent–child relation.

To avoid ill-formed fragment and event hierarchies, all fragments of a sequence diagram must fulfill a set of consistency properties, *s.t.* for any fragments $f_1, f_2 \in F$ with $f_1 \neq f_2$, $op(f_1) = (o_1, n_1)$ and $op(f_2) = (o_2, n_2)$ the following properties must hold:

1. no self–nesting: $f_1 \notin sub(f_1, x)$ for any $x \leq n_1$
2. no shared events (except if nested): $\forall x_1 \leq n_1, x_2 \leq n_2 : f_1 \notin sub(f_2, x_1) \wedge f_2 \notin sub(f_1, x_2) \implies ev(f_1, x_1) \cap ev(f_2, x_2) = \emptyset$
3. containment of events: $f_2 \in sub(f_1, x_1) \implies ev(f_1, x_1) \supseteq \bigcup_{n \leq n_2} ev(f_2, n)$

The first property avoids that any fragment may be nested in itself. The second property requires that any two disjunct fragments must not share events. The third property requires that if two fragments are nested the parent fragment must contain all events of its child fragments.

A sequence diagram can also contain a number of variables X and expressions Exp which are used on messages and fragments as guards. Apart from constant expressions on `loop` fragments, the variable and expression concepts are not relevant for our virtual integration analysis and are therefore ignored in the rest of this paper.

2.2 Test Case Sequence Diagrams

In our approach, we extend Definition 1 to *test case sequence diagrams* (see Definition 2), which are able to model the timed behavior of a test case. In a TCSD the object instances have two different roles. One instance represents the component of the system under test (SUT) for which the test case

is defined. All other object instances of the diagram represent test components. Test components are abstract components which will (in most cases) not appear within the real system architecture. Their only purpose is to provide input to and receive output from the ports of the SUT.

Therefore, we limit the set of events in messages, such that either the sending or the receiving event must be part of the *sut* object instance and the other event must be part of a test component. Self loops on the SUT are not allowed because the idea of the test components is to make the communication with the SUT visible, such that messages to the SUT can be interpreted as part of an input test vector and messages received from the SUT as part of the expected output.

TCSDs also allow the specification of timing constraints on events. To this end, the diagram type supports partition lines $\tau \in \mathscr{T}$ with $\tau = (e_1, \ldots, e_m, \delta)$ which are annotated with the timing information δ. A partition line cuts through all instance lines by introducing a new event in each line. Each TCSD has an implicit partition line τ_0 with $\delta = 0$ which indicates the start of the diagram. Every time stamp of following partition lines is relative to this initial line. The semantics are that every event before a partition line event has to happen before the annotated time stamp δ and every following event at least after δ time.

The consistency of partition lines is ensured by additional properties. The uniqueness property (1) requires that there are no two partition lines with the same time stamp. The completeness property (2) ensures that all instance lines have a partition line event separating their own events. The ordering property (3) ensures that the annotated times on all partition lines are in an ascending order. The last property (4) prevents the intersection of partition lines with fragments.

For the ordering of two partition lines we will write $\tau_1 < \tau_2$ as a short form for comparing the time steps $\delta_1 < \delta_2$.

To express timing constraints within a (sub-)fragment of the SUT instance line, a TCSD supports the concept of timeouts, which are represented by the set $\mathscr{C}$. Each timeout is a tuple $c = (e_1, e_2, \delta)$ consisting of two ordered events between which at most δ time units may pass. A timeout may be used within a subfragment, but both events of it must be part of the same fragment operand.

Definition 2 (Test Case Sequence Diagram)
A test case sequence diagram (TCSD) is a tuple $TC = (d, I, E, <, \Sigma_{msg}, M, F, X, Exp, sut, \mathscr{T}, \mathscr{C})$ *where:*

- $(d, I, E, <, \Sigma_{msg}, M, F, X, Exp)$ *is a sequence diagram;*
- $sut \in I$ *is the system under test instance line;*
- *for every* $m \in M$ *with* $m = (e_1, l, e_2) : (e_1 \in E_{sut} \wedge e_2 \notin E_{sut}) \vee (e_2 \in E_{sut} \wedge e_1 \notin E_{sut})$
- $\mathscr{T} \subseteq E^{|I|} \times \mathbb{N}_0$ *is the set of time partition lines,* s.t. *for every* $\tau_1, \tau_2 \in \mathscr{T}$ *with* $\tau_1 = (e_{11}, \ldots, e_{1m}, \delta_1)$ *and* $\tau_2 = (e_{21}, \ldots, e_{2m}, \delta_2)$*:*
  1. *uniqueness:* $\delta_1 = \delta_2 \implies \tau_1 = \tau_2$
  2. *completeness:* $\forall e_{1i}, e_{1j} : i \neq j \Leftrightarrow E_i \neq E_j$
  3. *ordering:* $\forall j : (\delta_1 < \delta_2) \Leftrightarrow (e_{1j}, e_{2j}) \in E_j \implies (e_{1j}, e_{2j}) \in <_j$;
  4. *no fragment cutting:* $\forall f_1 \in$ *with* $op(f_1) = (o_1, n_1), \forall n < n_1 : e_{11}, \ldots, e_{1m} \notin ev(f_1, n)$
- $\mathscr{C}$ *is a set of timeouts* $\mathscr{C} \subseteq E_{sut} \times E_{sut} \times \mathbb{N}$, s.t. *for every* $c = (e_1, e_2, \delta)$ *with* $e_1, e_2 \in E_{sut}$*:*
  1. *ordered:* $e_1 < e_2$
  2. *same fragment:* $\forall f \in F$ *with* $op(f) = (o_1, n_1) : e_1 \in ev(f, x) \Leftrightarrow e_2 \in ev(f, x)$ *for any* $x \leq n_1$

2.3 Timed-arc Petri Nets

In this section we will recall the formal definition of TAPN and introduce our notation which will be used in the rest of the document.

Definition 3 (Timed-arc Petri nets (based-on [3])
A timed-arc Petri net with transport arcs *(TAPN) is a tuple* $N = (P, T, R, c, R_{tarc}, c_{tarc}, \iota)$*, where:*

- P *is a finite set of places;*
- T *is a finite set of transitions* $(P \cap T = \emptyset)$*;*
- R *is the flow relation* $(R \subseteq (P \times T) \cup (T \times P))$*;*
- c *is a function that associates a time interval to each arc* (p,t) *in* R*,* s.t. $c : R|_{P\times T} \to \mathscr{I}$*;*
- $R_{tarc} \subseteq (P \times T \times P)$ *is the set of* transportation arcs *that satisfy for all* $(p,t,p') \in R_{tarc}$ *and all* $r \in P$*:* $((p,t,r) \in R_{tarc} \implies p' = r) \wedge ((r,t,p') \in R_{tarc} \implies p = r) \wedge (p,t) \notin R \wedge (t,p') \notin R$
- $c_{tarc} : R_{tarc} \to \mathscr{I}$ *associates a time interval to every transportation arc;*
- $\iota : P \to \mathscr{I}_{Inv}$ *assigns time intervals as invariants to places.*

The TAPN can be seen as a directed bipartite graph of separated places P and transitions T. The interconnection of the net is given by two flow relations. R defines arcs between places and transitions in both ways as known from conventional Petri nets, whereas R_{tarc} defines so-called transportation arcs. The main structural difference between those two kinds of arcs is that transportation arcs are always triples from a place over a transition to a place. The other arcs are separate tuple for arcs to a transition (p,t) or arcs from a transition (t,p). The additional condition on transportation arcs imposes, that there is at most one transportation arc between any two places.

In contrast to the original definition [3], we limit the supported time intervals $\mathscr{I}$ and $\mathscr{I}_{Inv}$ to be only closed $[\cdot,\cdot]$ or right-open $[\cdot,\cdot)$ intervals over $\mathbb{N}_0 \times (\mathbb{N}_0 \cup \{\infty\})$. Other kinds of intervals are not relevant for our virtual integration analysis. In addition, invariants are also not considered and we assume the default invariant $[0,\infty)$ for every place.

In the following we will use $p \xrightarrow{[\underline{t},\bar{t})} t$ as a short notation for the arc (p,t) with an associated time interval $[\underline{t},\bar{t})$ and $p \xrightarrow{[\underline{t},\bar{t})} t \longrightarrow p'$ for a sequence in a TAPN N where $(p,t),(t,p') \in R_N$ and $c_N(p,t) = [\underline{t},\bar{t})$. For transportation arcs we use the notation $p \xrightarrow{[\underline{t},\bar{t})}\!\!\blacklozenge\, t \longrightarrow\!\!\blacklozenge\, p'$ respectively.

The state of a Petri net is defined by its current placed tokens (marking). In a TAPN each token has an individual age which increases over time. Formally a marking M on a TAPN N is a function $M : P \to 2^{\mathbb{R}_0^+}$ which assigns every place $p \in P$ a set of positive real numbered tokens, *s.t.* each token $x \in M(p)$ of a marking fulfills the invariant of its assigned place: $x \in \iota(p)$. Only markings with a finite number of tokens are considered in this paper.

A marked TAPN is a tuple (N, M_0) of a TAPN N and a marking M_0 over the places of N. The dynamics are defined over changes of this marking, which follow the flow relations R and R_{tarc}. A transition t is said to be *enabled* if there exists at least one token in each of the places connected to its incoming arcs according to R for normal arcs and R_{tarc} in case of transportation arcs and these tokens are in the corresponding timing interval of c and c_{tarc} respectively. In addition, the tokens of normal arcs and transportation arcs have to fulfill the invariants of the target places. An enabled transition can *fire* by consuming exactly one token of matching age from each of its incoming arcs and producing one new token of each of its outgoing arcs. The age of these newly created token is either 0, if it is produced by a normal arc, or the age of the consumed token, if it is produced by a transportation arc. Instead of

firing a transition, a TAPN can perform a so called time delay in which the age of all token of the current marking is increased by the same timespan. The time delay is valid if all token still fulfill the invariants of their corresponding places. We write $M \xrightarrow{t} M'$ if the marking M' is reached from M by consuming t time units and firing enabled transitions of the TAPN. A marking M_k is said to be reachable from M_0 within k steps, if there is a sequence $M_i \xrightarrow{t_i} M_{i+1}$ for $0 \leq i \leq k$ and $i,k \in \mathbb{N}_0$.

For the construction of TAPNs from sequence diagrams, we need to be able to identify transitions with messages of the diagram. Therefore, we define a labeling function $\lambda_{trans} : T \to \Sigma_{trans}$, which associates each transition of a TAPN with a label from a labeling alphabet Σ_{trans}.

3 Virtual Integration of Sequence Diagrams

In the virtual integration scenario all TCSDs of the individual components are combined according to the interconnections of the ports provided by the system architecture. The procedure consists of three steps: First, the TCSDs are translated into TAPNs, which mimic the occurrences of the SUT events and impose the same timing constraints; Second, the individual TAPNs are combined to a single TAPN by synchronizing the communicated messages according to the architecture; And third, a consistency analysis is performed which checks if it is possible to successfully execute the combined TAPN.

3.1 Translation of TCSDs to TAPNs

For the translation of TCSDs to TAPNs we define a set of translation rules. The general idea is to construct a sequence of transitions (main sequence) in which each transition represents exactly one event of the SUT instance line. In the TAPN a single token is transported among this sequence and mimics the execution of the sequence diagram. The age of the token on the main sequence is restricted by timing constraints on the transitions according to the constraints on the partition line. Figure 2 shows the idea of the construction of the main sequence.

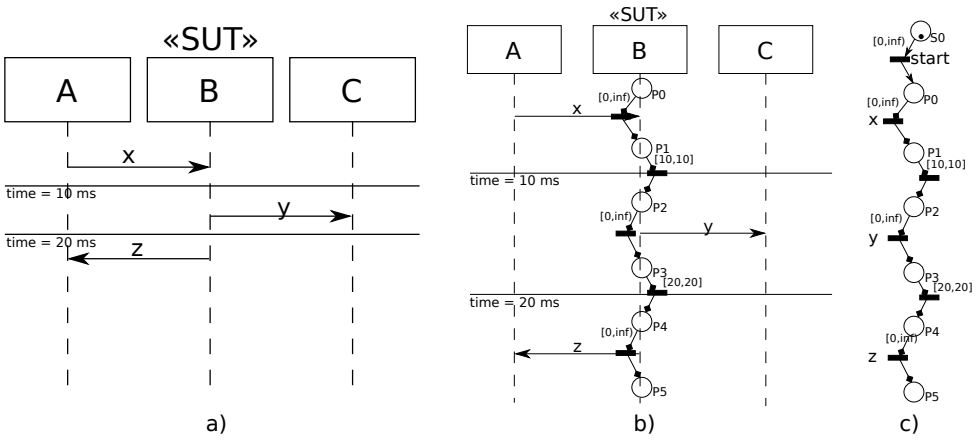


Figure 2: Example of the translation procedure of a TCSD to a TAPN. Subfigure *a)* shows a basic TCSD with the sequence of messages `x, y, z` and required timing-constraints. Subfigure *b)* shows how the message events relate to labeled transitions on the main transportation arc sequence of the TAPN. The timing-constraints are enforced by the interval guards. Subfigure *c)* shows the resulting marked TAPN including a place with a token and the delay transition at the beginning.

Messages are represented as labels on transitions. They don't add any additional semantic to the net, but are used for synchronization with other nets. Fragments and timeout events on the other hand, are translated by adding branches (one for each operand) to the main line. Each branch starts at the `enter` event transition and is merged again with the main line on the corresponding `exit` transition.

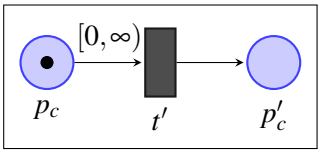
The formal construction is defined by a set of translation rules. Each rule applies to one kind of event. Their input on the one hand is the TCSD TC which has to be translated and an event $e_c \in E_{sut}$ of the SUT instance line which marks the current position in the diagram. On the other hand the other input is the so far partially generated TAPN PN and a place p_c to which the new Petri net elements have to be appended. The place p_c is either the last place of the generated main sequence or the end of the current branch, if e_c is part of a fragment operand. The output of the rule is an extended net PN' and the next event e'_c.

For the notation of the rules we will mark every new TAPN element added to PN with a prime. We will also use the function *next* to indicate the immediate following event within the SUT instance line according to the $<$ relation of the diagram. In case there is just a partial ordering because of multiple operands in a fragment, the rules will iterate the events of the operands sequentially. Intuitively, the transformation rule iterates over all events on the SUT instance line according to the graphical notation. If $next(e) = \emptyset$ then the event e is the last event on the instance line. In addition, the functions $first$ and $last$ are used to identify the first respectively the last event of a operand according to the ordering relation. Each rule consists of a formal description and a figure of the TAPN artifacts created by the rule. Places in *red* mark the attachment point to the prior constructed TAPN elements. Dashed lines indicate segments which have to be further extended.

The transformation process consists of a succeeding application of the different translation rules and an iterative construction of the corresponding TAPN. Initially, Translation Rule 1 is applied which creates the initial places and the marking M_0. Afterwards the Translation Rules 2–7 are applied according to the type of the current event e_c. They extend PN with a TAPN fragment modeling the sequence diagram event type until the last event has been handled ($next(e_c) = \emptyset$). Then the final Translation Rule 8 can be applied, which creates the marking M_{target} indicating the target state to be reached from the TAPN (PN, M_0).

Translation Rule 1 (Initial)

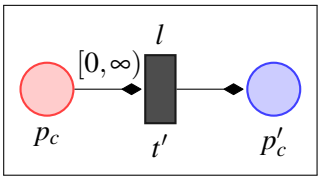
Let $e_c \in E_{sut}$ and PN be the empty TAPN. Create $p_c \xrightarrow{[0,\infty)} t' \longrightarrow p'_c$ with $\lambda'(t') = \varepsilon$, $e'_c = e_c$, and an initial marking M_0 consisting of a single token on place p_c with age 0.



Multiple sequence diagrams may be executed with an arbitrary time offset. Therefore, we prefix the main sequence with a transition, such that the TAPN can initially wait. The Translation Rule 1 adds a transition with normal arcs and no timing constraint before the first event of the TCSD. This models an arbitrary offset between the starting times of all TCSDs, *e.g.,* Figure 2 *c)* $S0 \xrightarrow{[0,\infty)} start \longrightarrow P0$.

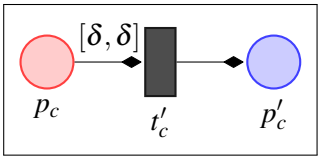
Translation Rule 2 (Send/Receive Message Event)

Let $e_c \in E_{sut}$ where $\exists m \in M$, s.t. $m = (e_c, l, e)$ or $m = (e, l, e_c)$ and $p_c \in P$. Append $p_c \xrightarrow{[0,\infty)} t' \longrightarrow p'_c$ with $\lambda'(t') = l$ and $e'_c = next(e_c)$



After the application of the initial rule, the other rules generate transportation arcs according to the events on the instance line. Translation Rule 2 models the sending and receiving of messages by adding a transportation arc to the main sequence with no additional timing constraints. The created transition t' is labeled with the label of the message. This label identifies the message (*e.g.,* it contains the port identifier and the sent/received content) and is later used for synchronization with other TAPNs, *e.g.,* Figure 1 $m = (e_{11}, l_1, e_{21})$.

Translation Rule 3 (Partition Line Event)
Let $e_c \in E_{sut}$ where $\exists \tau = (e_1, \ldots, e_n, e_c, e_m, \ldots, e_l, \delta)$ and $p_c \in P$. Append $p_c \xrightarrow{[\delta,\delta]} t'_c \longrightarrow p'_c$ with $\lambda'(t') = \varepsilon$ and $e'_c = next(e_c)$

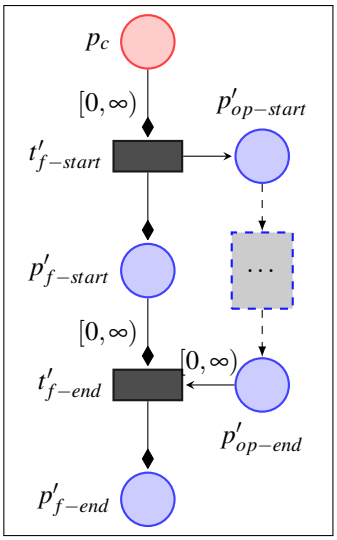


The timing constraints imposed by partition lines are also encoded by adding a single transportation arc to the main sequence of the graph. The interval $[\delta, \delta]$ limits the age of the main token to exactly δ in order to fire this transition. This represents the semantics of partition lines, *s.t.* all events before the partition line have to be executed before time stamp δ and all events after the partition line can only be executed after time stamp δ.

Translation Rule 4 (Par-/Alt-/Opt-Fragment Event)
Let $e_c \in E_{sut}$ where $\exists f \in F$ with $op(f) = (\Omega, N)$ with $\Omega \in \{\texttt{par}, \texttt{alt}, \texttt{opt}\}$ and e_c is `enter` event of f:
Append $p_c \xrightarrow{[0,\infty)} t'_{f-start} \longrightarrow p'_{f-start} \xrightarrow{[0,\infty)} t'_{f-end} \xrightarrow{[0,\infty)} p'_{f-end}$.
For each operand id $n < N$ do:

1. *Append $t'_{f-start} \longrightarrow p'_{op-start}$*
2. *Apply Rules 2–7 starting with $e_c = first(ev(f,n))$ and $p_c = p'_{op-start}$ until $e'_c \notin ev(f,n)$*
3. *Append $p'_{op-end} \xrightarrow{[0,\infty)} t'_{f-end}$ with $p'_{op-end} = p'_c$ after step 2.*

Finally, set $p'_c = p'_{f-end}$ and $e'_c = next(e_c)$.



Fragment blocks `alt, opt, par` [12] and the possible hierarchical structures thereof are encoded by adding additional paths, branching from the main sequence for each operand of the fragment. The main sequence is extended by two transitions. The first transition $t'_{f-start}$ represents the `enter` event of the fragment and the second t'_{f-end} the corresponding `exit` event. A new branch starting from $t'_{f-start}$ and ending at t'_{f-end} is added to the main sequence for each of the operands of the fragment. The branches itself are constructed according to the normal Translation Rules 2–7. This construction effectively reduces the `alt` and the `opt` fragments to the parallel execution construction of `par`. The motivation is that the resulting TAPN shall represent all possible execution paths of the original TCSD in order to provide all possible synchronization points for the virtual integration with other TAPN.

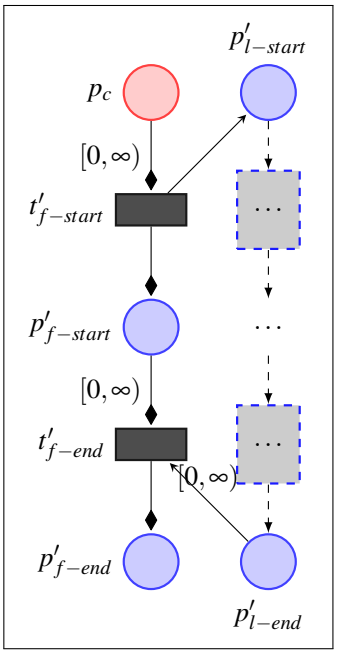
Translation Rule 5 (Strict-Fragment Event)
Let $e_c \in E_{sut}$ where $\exists f \in F$, $op(f) = (\Omega, N)$ with $\Omega =$ `strict` and e_c is `enter` event of f: Set $e'_c = next(e_c)$

Translation Rule 5 handles `strict` fragments [12], which is the default semantic in our construction. Therefore, it does not require any additional handling and the corresponding `enter` and `exit` events can be skipped.

Translation Rule 6 (Loop-Fragment Event)
Let $e_c \in E_{sut}$ *where* $\exists f \in F,\ op(f) = (\Omega, 1)$ *with* $\Omega =$ `loop`, $guard(f) = N$ *with* $N \in \mathbb{N}$, *and* e_c *is* `enter` *event of* f*:*
Append $p_c \xrightarrow{[0,\infty)} t'_{f-start} \longrightarrow p'_{f-start} \xrightarrow{[0,\infty)} t'_{f-end} \xrightarrow{[0,\infty)} p'_{f-end}$.

1. *Append* $t'_{f-start} \longrightarrow p'_{l-start}$
2. *Repeat N-times:*
   - *Apply Rules 2–7 starting with* $e_c = first(ev(f,1))$ *and* $p_c = p'_{l-start}$ *until* $e'_c \notin ev(f,n)$
3. *Append* $p'_{l-end} \xrightarrow{[0,\infty)} t'_{f-end}$ *with* $p'_{l-end} = p'_c$ *after step 2.*

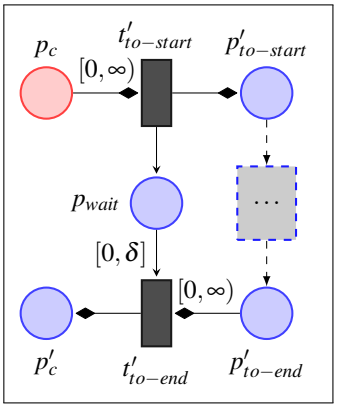
Finally, set $p'_c = p'_{f-end}$ *and* $e'_c = next(e_c)$.



The `loop` fragment is only supported if its guarding expression is a constant number. The construction described in Rule 6 is similar to the construction used for the `par` fragments. Except it is limited to a single operand and instead of adding multiple branches, it extends the looping branch starting with $p'_{l-start}$ N-times where N is the number of unrollings of the loop.

Translation Rule 7 (Timeout Event)
Let $e_c \in E_{sut}$ *where* $\exists c = (e_{start}, e_{end}, \delta) \in \mathscr{C}$ *with* $e_c = e_{start}$*:*

1. *Append* $p_c \xrightarrow{[0,\infty)} t'_{to-start} \longrightarrow p'_{to-start}$
2. *Apply Rules 2–7 starting with* $e_c = next(e_c)$ *and* $p_c = p'_{to-start}$ *until* $e'_c = e_{end}$
3. *Append* $p'_{to-end} \xrightarrow{[0,\infty)} t'_{to-end} \longrightarrow p'_c$ *with* $p'_{to-end} = p'_c$ *after step 2.*
4. *Append* $t'_{to-start} \longrightarrow p_{wait} \xrightarrow{[0,\delta]} t'_{to-end}$



Timeout events define a timing constraint between the occurrences of the start event e_{start} and the end event e_{end}, *e.g.*, the timeout operator $(e_{25}, e_{29}, 5ms)$ in Figure 1 constraints the time between the occurences of the events e_{26}, e_{27}, e_{28} to $5ms$. The construction described in Translation Rule 7 creates a new transportation arc on the main sequence for each of these events and a connected place with the corresponding timing constraint as interval. In contrast to fragment event rules, this rules constructs all events between e_{start} and e_{end} directly on the main sequence rather than on a branch.

Translation Rule 8 (Termination)
Let $e_c \in E_{sut}$ of SD and $p_c \in P$ of PN. If $next(e_c) = \emptyset$ then PN is the transformed TAPN of SD with correspondence in timed reachability of the marked TAPN (PN, M_0) to a marking M_{target}, where M_0 is the initial constructed marking and M_{target} is a marking consisting of a single token (with arbitrary age) on p_c.

The translation process terminates with the application of Translation Rule 8. This rule is only applicable if the end of the SUT instance line is reached ($next(e_c) = \emptyset$) and it creates the target Marking M_{target} representing the state after the execution of the TCSD in the constructed TAPN.

3.2 Synchronization of TAPN

The virtual integration is done according to a system architecture A, which instantiates components and specifies how they are interconnected. Each component is associated with its own set of test cases, specified as TCSDs. Within a TCSD the component itself is identified as the SUT instance line. The other instance lines are test components, which represent virtual communication partners for the ports of the component.

Given the connection between the ports of the components within the architecture it is possible to create a mapping of all test components to existing components of the architecture. This mapping directly corresponds to a mapping of SUT instance lines and test component instance lines.

We write $i_1 \cong_A i_2$ for two instance lines $i_1 \in I_1$ and $i_2 \in I_2$ to indicate that i_1 and i_2 are in a mapping relation according to the interconnections of the architecture A.

The synchronization of these instance lines is done on the basis of the messages transmitted on the common ports. For the compatibility of messages we define a similar relation in Definition 4. Two messages are compatible if their source and target instance lines are in the mapping relation $\cong_A$.

Definition 4 (Compatibility of Messages)
Let TC_1, TC_2 be two TCSD with $TC_1 \neq TC_2$ and A be an architecture connecting the components: The compatibility $\cong_A$ of two messages $m_1 = (e_{11}, l_1, e_{12}) \in M_{TC_1}$ with $e_{11} \in E_{i_{11}}, e_{12} \in E_{i_{12}}$ and $m_2 = (e_{21}, l_2, e_{22}) \in M_{TC_2}$ with $e_{21} \in E_{i_{21}}, e_{22} \in E_{i_{22}}$ is defined as: $(m_1 \cong_A m_2) := (i_{11} \cong_A i_{21}) \wedge (l_1 = l_2) \wedge (i_{12} \cong_A i_{22})$

Given this definition of compatibility, we can combine multiple TCSDs by synchronizing all compatible messages of the two diagrams. This synchronization process is in general not unique *e.g.,* if there are multiple instances of messages with the same label). In this case all combinations of potential synchronization points have to be considered. To this end, we transform each individual TCSD into its corresponding TAPN. In the TAPN representation, each message is represented by a unique transition (see Translation Rule 2). In case of a synchronization, the two transitions representing the compatible messages can be combined to a single new transition as depicted in Figure 3. Formally, all sets and functions of the two TAPN are merged, new transitions are introduced replacing the individual synchronized transitions, and the arcs are redirected to the new transitions.

3.3 Consistency Analysis

After all individual TAPNs are merged to a single net, the final step of the virtual integration analysis is to determine if this TAPN is consistent. Consistency in this case means that the target marking of the

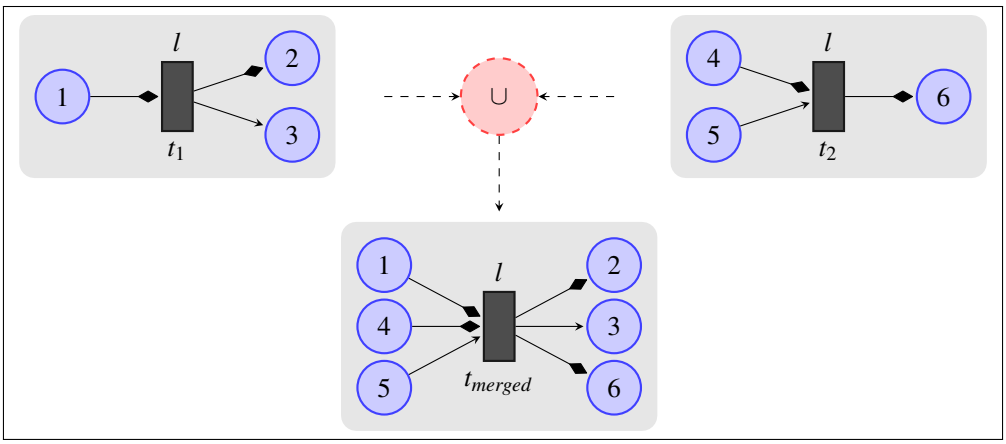


Figure 3: Example of the combination of two (partial) TAPNs. The two transitions t_1 and t_2 of the nets on the top are synchronized by replacing the transitions with a newly created transition t_{merged}. All arcs (transportation and normal) to t_1 or t_2 are redirected to t_{merged}. The places in this figure are numbered to identify them in the merged TAPN.

combined TAPN (the union of all individual target markings) is reachable from its initial marking (the union of all initial markings).

This procedure detects timing constraint violations and ordering inconsistencies of messages. Timing constraints in the constructed TAPN relate to interval bounds (lower and upper) on the age of the token on the individual main sequences and the token of timeout operator branches (see Translation Rule 7). If two or more nets have to synchronize on a common transition the ages of these token must still fulfill their guards even if the token have to wait additional time for the synchronization. Formally, the analysis has to check if all interval constraints on synchronized transitions have a non-empty intersection. This ensures that at least one successful execution of the integrated components fulfills all timing constraints.

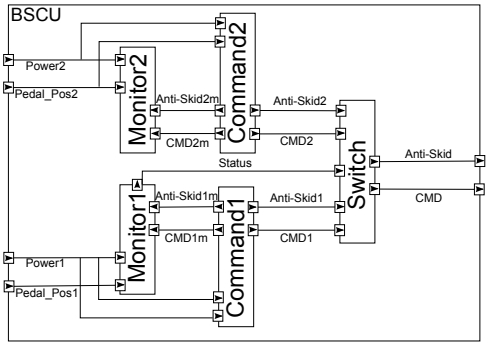
The second detectable kind of inconsistency is the problem regarding the ordering of messages. If for example one test case defines a strict ordering of two messages m_1 and m_2 and a second test case specifies the opposite ordering of first m_2 and then m_1, the merged TAPN can never reach its target marking. The messages m_1/m_2 in both test cases are translated into transitions t_1/t_2 according to Transition Rule 2. After synchronizing the transitions, the net has a classical deadlock in its transitions: t_1 can by construction only fire after t_2 fired, but t_2 has to wait until t_1 fired.

In case there are multiple possibilities to synchronize two nets, the analysis has to consider all possible points of synchronization. For our analysis we consider it as sufficient, if at least one of the combinations succeeds the reachability analysis. Alternatively, one could impose a stronger consistency concept if it is required that all possible combinations pass the analysis.

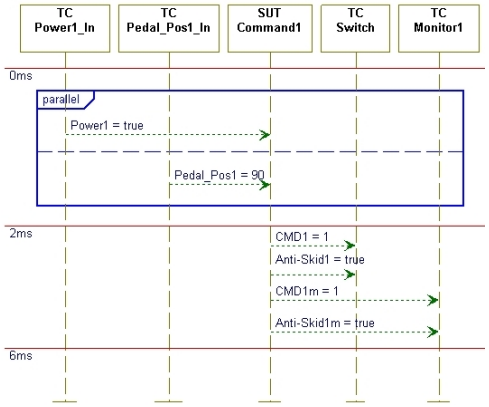
In general test cases define only the fragment of the total component behavior which is relevant for the test case. Therefore, the specified message sequences are in most cases incomplete. This can lead to a *false* inconsistent result of the analysis. The prior mentioned inconsistent sequence m_1 then m_2 may be consistent if we assume that the second test case just neglected a first occurrence of m_1 *e.g.,* the full sequence would be m_1, m_2, m_1. Therefore, inconsistency results may just be cases of underspecified test cases.

4 Evaluation

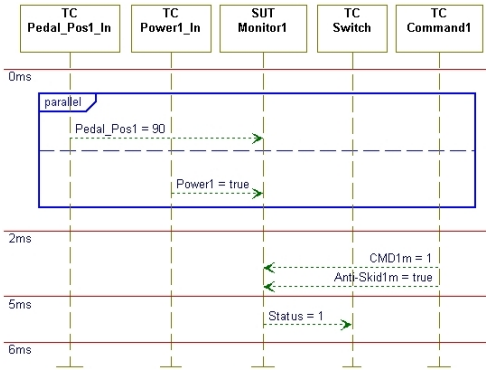
We demonstrate the virtual integration analysis presented in this paper on an example that is complex enough for our purpose but still has an acceptable degree of simplicity to understand the context. The Brake System Control Unit (BSCU) is part of a Wheel Braking System (WBS) example that was used to describe the safety assessment for certification of civil aircraft in the SAE standard ARP4761 [1]. The architecture of the BSCU, which consists of two redundant subsystems, is shown in Figure 4a.



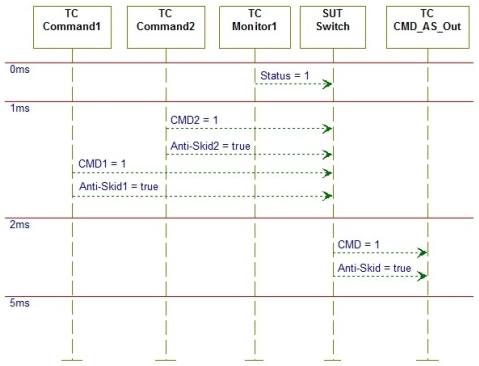
(a) Architecture



(b) Test case with SUT `Command1`



(c) Test case with SUT `Monitor1`



(d) Test case with SUT `Switch`

Figure 4: Subfigure a) shows the component model of the BSCU. Subfigures b) - d) depict test cases used for the demonstration of our approach. The SUTs and test components of each test case are mapped on the corresponding component in the shown architecture.

We created a set of test cases to demonstrate our approach. These test cases represent the interaction of specific components of the BSCU, *e.g.,* a test case for the *Command* unit (Figure 4b), *Monitor* (Figure 4c), and the *Switch* (Figure 4d). Applying our approach, each of these sequence diagrams is translated into one timed-arc Petri net. The three resulting Petri nets are then being merged into one single TAPN, see Figure 5.

The translation rules introduced in section 3.1 can be identified in Figure 5. For example a translation of a partition line (Translation Rule 3) can be seen in the top left corner ($P20 \xrightarrow{[0,0]} t \longrightarrow P21$). Another example is a realisation of the *par*-operator (Translation Rule 4). Its main sequence is represented by

$P21 \xrightarrow{[0,\infty)} t_1 \longrightarrow P24 \xrightarrow{[0,\infty)} t_2 \longrightarrow P27$ and its branches are starting at the places $P22$ and $P23$.

Performing the consistency analysis means that the target marking M = (End_{TCSD_Mon},$End_{TCSD_{Com}}$, $End_{TCSD_{S}witch}$) is reachable. In this case, the target marking is not reachable because of a deadlock, caused by the transitions *Status*, *CMD1m*, *AntiSkid1m*, *CMD1* and *AntiSkid1*. Since this is detected by our analysis, we are able to fix this error before the integration of the real system.

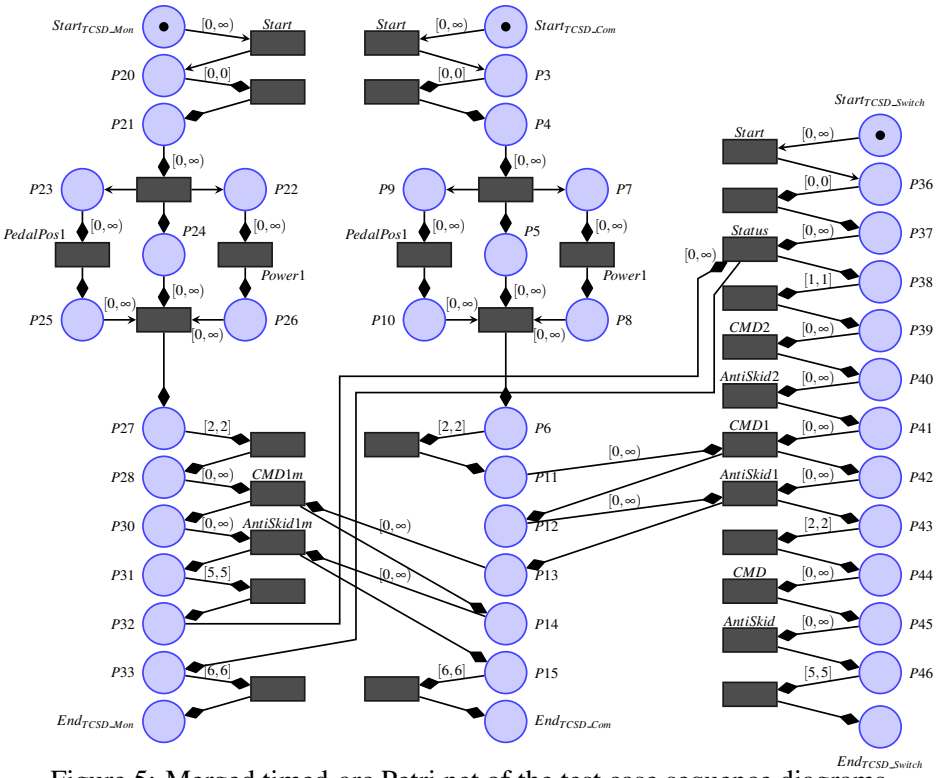


Figure 5: Merged timed-arc Petri net of the test case sequence diagrams.

5 Conclusion

We presented an approach to analyze sequence diagram-based test cases. Therefore, a concept of test case sequence diagrams was introduced, which allows to annotate timing information to test cases. In order to formalize these test cases we extended the sequence diagram formalism of Bowles by the additional timing information. In addition, we adapted the Petri net formalism of Byg to represent the needed test case elements. For the translation of a TCSD into a TAPN, a set of translation rules was presented. These TAPNs can be merged into one single Petri net. On the basis of the merged Petri net, we were able to analyse if a set of test cases is consistent in the sense of ordering and timing behavior.

The applicability of the approach was demonstrated using a example from the ARP 4761. We used IBM Rational Rhapsody to specify test cases for this BSCU and developed a prototype to extract the sequence diagram-based specifications. It also translates them into a TAPN to enable the analysis.

In the future we want to evaluate this approach on a larger scale design process, in which our TCSDs are used for requirements specification as well as for test cases. In addition, we want to extend the scope of the analysis to enable support for life sequence charts [5]. This will enable the integration of the anal-

ysis into the early stages of the development process, *e.g.,* to analyse requirements for early verification. Other forms of synchronization of TAPNs, *e.g.,* check for time interval inclusion in request/response scenarios are also planned.

Acknowledgments The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement n°269335. It was also partially funded by the German Federal Ministry of Education and Research (BMBF), grant "SPES XT, 01IS12005M" .

References

[1] SAE ARP4761 (1996): *Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment*. SAE International, pp. 1–331.

[2] Juliana Bowles & Dulani Meedeniya (2010): *Formal Transformation from Sequence Diagrams to Coloured Petri Nets*. *2010 Asia Pacific Software Engineering Conference*, pp. 216–225, doi:10.1109/APSEC.2010.33.

[3] Joakim Byg & Kenneth Yrke Jørgensen (2009): *An Efficient Translation of Timed-Arc Petri Nets to Networks of Timed Automata*. *Formal Methods and Software Engineering* 5885(1), pp. 698–716, doi:10.1007/978-3-642-10373-5_36.

[4] Joakim Byg, Kenneth Yrke Jørgensen & Jirí Srba (2009): *TAPAAL: Editor, simulator and verifier of timed-arc Petri nets*. In: *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, pp. 84–89, doi:10.1007/978-3-642-04761-9_7.

[5] Werner Damm, Andreas Baumgart, Eckard Böde, Matthias Büker, Tayfun Gezgin, Stefan Henkler, Hardi Hungar, Bernhard Josko, Markus Oertel, Thomas Peikenkamp, Philipp Reinkemeier, Ingo Stierand & Raphael Weber (2011): *Architecture Modeling*. Technical Report, OFFIS, Oldenburg.

[6] Christoph Eichner, Hans Fleischhack & Roland Meyer (2005): *Compositional semantics for UML 2.0 sequence diagrams using Petri nets*. In: *In 12th Int. SDL Forum, volume 3530 of LNCS*, pp. 133–148, doi:10.1007/11506843_9.

[7] HM Hanisch (1993): *Analysis of place/transition nets with timed arcs and its application to batch process control*. *Application and Theory of Petri Nets 1993*, pp. 282–299, doi:10.1007/3-540-56863-8_52.

[8] ISO (2009): *ISO/DIS 26262-1 - Road vehicles "Functional safety" Part 1 Glossary*. Technical Report.

[9] Xiaoshan Li, Zhiming Liu & He Jifeng (2004): *A formal semantics of UML sequence diagram*. In: *Australian Software Engineering Conference Proceedings*, 292, pp. 168–177, doi:10.1109/ASWEC.2004.1290469.

[10] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong & Z. Guoliang (2004): *Generating test cases from UML activity diagram based on gray-box method*. In: *Software Engineering Conference, 2004. 11th Asia-Pacific*, 60233020, pp. 284–291, doi:10.1109/APSEC.2004.55.

[11] Zoltán Micskei & Hélène Waeselynck (2010): *The many meanings of UML 2 Sequence Diagrams: a survey*. *Software & Systems Modeling* 10(4), pp. 489–514, doi:10.1007/s10270-010-0157-9.

[12] OMG (2010): *OMG Unified Modeling Language TM (OMG UML), Superstructure v2. 3 . 2010.* Technical Report May.

[13] Sven Sieverding (2011): *Sequenzdiagrammbasierte Test- und Analysemethoden von AUTOSAR-Softwarekomponenten (SWCs)*. Master thesis, Oldenburg.

[14] Dehla Sokenou (2006): *Generating test sequences from UML sequence diagrams and state diagrams*. *Informatik 2006: Informatik für Menschen* 2(94), pp. 236–240.

[15] Jirí Srba (2005): *Timed-arc Petri nets vs. networks of timed automata*. In: *Applications and Theory of Petri Nets*, pp. 1273–1278, doi:10.1007/11494744_22.

[16] Jirí Srba (2008): *Comparing the expressiveness of timed automata and timed extensions of Petri nets*. *Formal Modeling and Analysis of Timed Systems*, pp. 15–32, doi:10.1007/978-3-540-85778-5_3.