Extending a Component Specification Language with Time

Björn Metzler and Heike Wehrheim

Institut für Informatik
Universität Paderborn
33098 Paderborn, Germany

Abstract
In a formal approach to component specification, interfaces are usually described using pre- and postconditions of methods or protocols. In this paper we present an approach for integrating time into a component specification language which already allows for pre/post and protocol descriptions. The specification of timing aspects is indispensable when treating components of embedded systems underlying hard real-time requirements. In order to allow for a smooth integration into the existing specification language and to ease reading and writing of interfaces, we do not extend the language with yet another formalism (for time), but instead only add a specific feature (i.e. clocks) to it. We define a semantics for this new specification language in terms of timed automata, which thus also opens the possibility of analysing interface descriptions with the UPPAAL model checker. Furthermore we give timed simulation conditions and prove their soundness with respect to inclusion of timed traces, the notion of implementation in timed automata. This implementation relation can be used as a correctness criterion for interoperability and substitutability checks.

Key words: Interface specification, timed automata, pre/post conditions, protocols, simulation, verification.

1 Introduction
Interfaces of components are typically described by giving signature lists, pre- and postconditions of methods or by defining protocols (i.e. valid call sequences). Different approaches and languages have been proposed for these purposes: the signature list only technique is the approach adopted by most industrial middleware platforms, pre- and postconditions are for instance used

1 Email: bmetzler@uni-paderborn.de
2 Email: wehrheim@uni-paderborn.de

©2006 Published by Elsevier Science B. V.
in [16,28,18] and protocol definitions for components given as finite state automatata, process algebra descriptions or temporal logic can be found in [20,21,11]. For embedded systems, it is however also important to specify timing constraints of interfaces, such as deadlines guaranteed or expected by a component.

In this paper we set out to develop a component specification language which allows for the specification of pre- and postconditions, protocols and timing constraints. The starting point here is an already existing language which contains features for specifying pre- and postconditions and protocols. The notation, called CSP-OZ [9], is a combination of the process algebra CSP [12,22] with the state-based, object-oriented formalism Object-Z [24]. The process algebra is used to specify specific call sequences guaranteed/expected by the component, the state-based formalism handles data-dependent aspects like pre- and postconditions of methods. Both formalisms come with built-in notions of refinement which is the formal development concept guaranteeing substitutability. Thus refinement can be used as correctness criterion for interoperability and substitutability checks. The integrated notation CSP-OZ is now extended to allow for the specification of timing constraints. To this end, we however do not integrate a third formalism into the existing combination but instead only add a specific clock type for Object-Z variables. Clock variables can be declared, queried and changed just like ordinary variables. These clock variables allow for the specification of deadlines, minimum and maximum delays between method calls etc.. This is similar to the way finite automata are extended to timed automata [1], which is the standard formalism for describing systems with timing aspects (they, however, do not allow for a high level description of state-based and behavioural aspects).

For this specification language (called timed CSP-OZ) we furthermore propose a method for analysing component interfaces and we define a formal notion of implementation, which can - like refinement - be used for substitutability checks. The analysis method is based on a semantics for the language in terms of timed automata (or more precisely, timed transition systems, since the semantics will not always yield a finite state automaton). In case of a finite number of states we can then use one of the timed automata model checkers for verification (e.g. Kronos [27] or UPPAAL [3]).

Based on this semantics we can furthermore use the notion of implementation associated with timed automata for timed CSP-OZ. The implementation relation for timed automata is inclusion of timed traces (language inclusion for words with time stamps). We define timed simulation conditions and show their soundness with respect to this relation. This opens the way for a stepwise proof of implementation.

The paper is structured as follows. Next, we start with a simple example of a timed CSP-OZ specification on which we explain the general idea and which will serve as an illustration of the main results in the next sections. Section 3 gives a short introduction to timed automata. We then define the semantics
for timed CSP-OZ specifications in terms of timed automata. In Section 4 we show how to analyse interface specifications in timed CSP-OZ with the timed automata model checker UPPAAL. Section 5 gives timed simulation conditions which can be used to prove language inclusion relationships between interfaces (and thus substitutability). The last section concludes and discusses related work.

2 A first example

The formalism timed CSP-OZ that we introduce in this paper is an extension of CSP-OZ with time. CSP-OZ [9] is a combination of the process algebra CSP [12] and the state-based specification formalism Object-Z [24]. It employs CSP to describe aspects of dynamical behavior of components (allowed call sequences of methods); it uses Object-Z to describe data aspects, i.e. the static behavior of operations like pre- and postconditions. For this, Object-Z uses set theory and predicate logic.

We directly give a timed CSP-OZ specification here since these specifications will in general not look very different from plain CSP-OZ specifications. For timed CSP-OZ we always assume a type Clock taking values from the set of non-negative reals (the time):

$$\text{Clock} == \mathbb{R}_+$$

Variables of type clock can be declared as attributes of classes and may (under some restrictions) appear in predicates within method schemas. The following example shows an abstract specification of the interface of a watchdog component. A watchdog component should control a certain method note which is to be repeatedly executed within 10 time units after its last occurrence. An alarm can either be raised by a ring after the expiration of the deadline or after at least 8 seconds by using a flash signal. A component implementing this interface may choose one of these options. Below, the component is specified as an Object-Z class.

$$\begin{align*}
\text{Watchdog} \\
\text{method} \text{ note}, \text{ ring}, \text{ flash} \\
\text{main} = \text{ note} \rightarrow \text{ main} \\
\quad \quad \quad \Box (\text{ ring} \rightarrow \text{ Alarm}_r, \Box \text{ flash} \rightarrow \text{ Alarm}_f) \\
\text{Alarm}_r = \text{ ring} \rightarrow \text{ Alarm}_r, \\
\text{Alarm}_f = \text{ flash} \rightarrow \text{ Alarm}_f
\end{align*}$$

There are slight differences in the use of Object-Z within CSP-OZ and in the standard definition. In this paper we will, nevertheless, plainly say Object-Z even when meaning the Object-Z part of CSP-OZ.
The class first consists of an enumeration of the set of methods it supplies and uses (usually with their signatures, which are, however, empty in this simple example). Next, a set of CSP process equations (with main process main) gives the protocol of the component. Finally, a number of Z schemas describe the state space, the initialisation and the methods of the class: The watchdog class has three methods note, ring and flash which have neither input nor output parameters. The CSP part specifies that note can be repeatedly executed until either the first ring or the first flash happens. Afterwards, only further ring’s resp. flash’s can follow (□ is the CSP operator for external choice and → the prefix operator describing sequencing). The Object-Z part declares three attributes: alarm for describing that an alarm has been raised and two clocks \(x_r\) and \(x_f\) used for determining whether the timing requirements are met. The class invariant specifies a condition which relates alarm to the clock variable \(x_r\). The three operation schemas define the execution of methods note, ring and flash: the precondition of note is the clock \(x_r\) being less than ten\(^4\), the postcondition specifies both clocks to be zero (\(x'_r\) denotes the value of \(x_r\) in the after state). Method ring can be executed if clock \(x_r\) is greater or equal to 10 and upon execution the alarm is set. The same holds for method flash which may be executed if clock \(x_f\) is greater or equal to 8.

After having introduced timed CSP-OZ by means of a simple interface specification, we are next interested in the analysis of such specifications and in the definition (and the checking) of an implementation relation between specifications. Such a relation could be used for substitutability checks. To

\(^4\) Like Object-Z and CSP-OZ, we use a blocking semantics for operations here: the precondition acts as a guard to the method execution.
this end, we will first define a semantics for timed CSP-OZ specifications.

3 Semantics

Protocols of components are quite often described by finite state automata. Here, we choose a similar formalism for our semantics, namely an extension of finite automata to time, called timed automata \cite{1,4}. The timed automaton for a timed CSP-OZ specification will capture the complete behaviour of the specification, including the data dependent aspects covered by Object-Z. Timed automata can, however, not be used to specify pre- and postconditions of methods, thus we do not directly use timed automata for interface specifications, only for their semantics.

3.1 Timed automata

Timed automata are finite automata enhanced with clock variables which can be queried and reset on transitions. To ensure decidability of the emptiness problem (and thus allow for verification), the conditions on clocks are usually restricted. We will later fix similar restrictions on our Z predicates over clocks to ensure that timed CSP-OZ can be safely mapped onto timed automata.

Definition 3.1 Let $X$ be a set of clock variables. The set of clock conditions over $X$, $\Phi(X)$, is given by the following grammar (where $c \in \mathbb{Q}_{\geq 0}$):

$$\varphi ::= x = c \mid x \leq c \mid c \leq x \mid x < c \mid c < x \mid (\varphi \land \varphi)$$

We let $\Sigma$ describe the global alphabet of operations of a specification, which we will call events, $\Sigma^*$ the set of finite words, $\Sigma^\omega$ the set of infinite words over $\Sigma$, $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ and $X$ a global set of clock variables. Since automata always have a finite set of states but our specifications are easily infinite state (due to data) we first define timed (transition) systems as ”timed automata with infinite number of locations” and afterwards have timed automata as a special case of timed systems. A timed system has all the ingredients of a finite state machine (i.e. states or locations, transitions and an initial state), and in addition has a labelling of transitions with clock conditions (determining when the transition can be taken) and sets of clocks (giving all clocks reset upon taking the transition). Furthermore, clock conditions can be associated with locations meaning that the automaton can only be in this location when the clock condition holds.

Definition 3.2 A timed system is a tuple $T = (Q, \rightarrow, q_0, I)$ where

- $Q$ is a (possibly infinite) set of locations,
- $\rightarrow \subseteq Q \times \Sigma \times \Phi(X) \times 2^X \times Q$ are the transitions (or edges),
- $q_0$ is the initial location,
- $I : Q \rightarrow \Phi(X)$ assigns invariants to locations.
We write $q \xrightarrow{a,\varphi,Y} q'$ for $(q, a, \varphi, Y, q') \in \rightarrow$; the clock condition $\varphi$ will sometimes also be called the guard of the transition, $a$ its label and $Y$ its resets. In case of $Q$ being a finite set we say that $T$ is a timed automaton.

Note that unlike Alur and Dill's timed automata we have no Büchi acceptance states here, instead progress is achieved by attaching invariants to states. This is a variant of timed automata (used in UPPAAL and Kronos as well) which is sometimes also referred to as timed safety automata.

Like finite state automata, timed automata accept languages. In this case, however, languages are sets of timed words (or timed traces). In contrast to a language over the alphabet $\Sigma$ consisting of $\sigma \in \Sigma^\infty$, a timed language is a set of tuples $(\sigma, \tau) \in \Sigma^\infty \times \mathbb{R}_+^\infty$:

**Definition 3.3** A timed trace $(\sigma, \tau) = (a_1 a_2 \ldots, \tau_1 \tau_2 \ldots)$ is a pair of finite or infinite sequences of events $a_i \in \Sigma$ and time values $\tau_i \in \mathbb{R}_+$ such that $\tau_{i+1} \geq \tau_i$ for all $i \geq 1$. In case of $\sigma$ and $\tau$ being finite both sequences furthermore have the same length, i.e. $\#\sigma = \#\tau$.

An example of a timed trace of our Watchdog specification is

(note note ring ring ring ... 5.3 14.9 24.9 27.4 33.8 ...)

The set of timed traces of a timed system can be derived by looking at its possible configurations. A configuration $\langle q, \nu \rangle$ consists of a location $q$ and a clock valuation $\nu : X \rightarrow \mathbb{R}_+$. We define two operations on clock valuations which are used to describe the executions of a timed system:

- **time shift:**

  $$(\nu + d)(x) = \nu(x) + d$$

- **modification:**

  $$\nu[Y := d](x) = \begin{cases} 
  d, & \text{if } x \in Y, \\
  \nu(x), & \text{else.}
  \end{cases}$$

A timed system is able to perform two kinds of transitions, delay transitions, where the time advances while the automaton stays in a location

$$\langle q, \nu \rangle \xrightarrow{d} \langle q, \nu + d \rangle \text{ iff } \nu \models I(q) \land \nu + d \models I(q), d \in \mathbb{R}_+$$

and action transitions, where an event-labelled transition is taken

$$\langle q, \nu \rangle \xrightarrow{a,\varphi,Y} \langle q', \nu' \rangle \text{ iff } q \xrightarrow{a,\varphi,Y} q' \land \nu \models \varphi, \nu' = \nu[Y := 0],$$

$$\nu \models I(q), \nu' \models I(q')$$

A timed trace $(\sigma, \tau) = (a_1 a_2 \ldots, \tau_1 \tau_2 \ldots)$ is in the language $L(T)$ of a timed
system \( T \) iff there is an execution

\[
\langle q_0, \nu_0 \rangle \xrightarrow{d_1} \langle q_1, \nu_1 \rangle \xrightarrow{a_1} \langle q_2, \nu_2 \rangle \rightarrow \ldots
\]

where \( \tau_i = \tau_{i-1} + d_i, \tau_0 := 0 \) and \( \nu_0(x) = 0 \) for all clock variables \( x \). Similar to the untimed case, the language of a timed automaton can be used to define an implementation relation between timed automata. A timed system \( T_2 \) is said to be an implementation of \( T_1 \) iff \( L(T_2) \subseteq L(T_1) \).

The timed automaton for a timed CSP-OZ specification will be constructed in two steps: First, we derive a timed automaton for the CSP part and the Object-Z part alone. In the second step, these will be combined via parallel composition giving rise to an automaton which obeys the restrictions of both parts. Hence we next define a parallel composition operator \( ||_A \) on timed automata, describing synchronous parallel composition requiring synchronisation on all actions in the set \( A \subseteq \Sigma \):

**Definition 3.4** Let \( T_i = (Q_i, \rightarrow_i, q_{0,i}, I_i), i = 1, 2, \) be timed systems over \( \Sigma_1 \) and \( \Sigma_2 \) with disjoint set of clock variables \( X_1, X_2 \), and let \( A = \Sigma_1 \cap \Sigma_2 \) be the synchronisation set. The **synchronous parallel composition** of \( T_1 \) and \( T_2 \), \( T_1 ||_A T_2 \), is defined to be the timed system \( T = (Q, \rightarrow, q_0, I) \) such that

1. \( Q = Q_1 \times Q_2 \),
2. \( (q_1, q_2) \xrightarrow{a, \phi, Y} (q'_1, q'_2) \) iff
   - \( a \in \Sigma_1 \cap \Sigma_2 \) and
     - \( q_i \xrightarrow{a, \phi_i, Y_i} q'_i, i = 1, 2, \) and \( \phi = \phi_1 \land \phi_2, \) \( Y = Y_1 \cup Y_2 \) (joint transition), or
   - \( a \in (\Sigma_1 \setminus A) \) and \( q_1 \xrightarrow{a, \phi_1, Y_1} q'_1, q'_2 = q_2, \) and
   - \( a \in (\Sigma_2 \setminus A) \) and \( q_2 \xrightarrow{a, \phi_2, Y_2} q'_2, q'_1 = q_1, \)
3. \( q_0 = (q_{0,1}, q_{0,2}) \),
4. \( I(q_1, q_2) = I_1(q_1) \land I_2(q_2) \).

This now allows us to give a semantics to a timed extension of CSP-OZ.

3.2 **Semantics of timed CSP-OZ**

First, we have to precisely specify what kinds of predicates over clock variables are allowed in specifications. As already mentioned, we assume to have a type \( \text{Clock} \) given with values from the nonnegative reals \( \mathbb{R}_+ \). Every timed CSP-OZ specification may contain a number of clock variables \( x_1, \ldots, x_n \) and a number of variables \( v_1, \ldots, v_m \) of other types. We then impose the following restrictions on the use of clock variables:

- the init schema specifies all clocks to be initially zero (and furthermore to uniquely fix values for the other variables, in order to have a unique initial state\(^5\));

\(^5\) This restriction can easily be lifted.
• operation schemas may contain clock conditions over unprimed clock variables plus predicates of the form \( x_i' = 0 \) (since in terms of modifications of timed automata, clocks may only be queried and reset);

• we need to ensure that the state schema must not have predicates over clock variables other than of the following form:

\[
    p \Rightarrow \varphi
\]

where \( \varphi \) is a clock condition and \( p \) a predicate over variables \( v_1, \ldots, v_m \). This restriction is necessary for a unique assignment of invariants to locations of the timed automata.

These conditions ensure that timed CSP-OZ specifications can be mapped onto timed automata. For defining the semantics we next have to separate the clockless parts of the specification from those with clocks. We define \( cl(schema) \) to be the clockless part of a schema, i.e. the declarations and predicates over non-clock variables, \( cc(schema) \) to be the clock part of a method schema, \( cinv(schema) \) to be the predicate(s) of the state schema relating clock variables and other variables, and \( reset(schema) \) to be the set of clocks \( x_i \) with predicates \( x_i' = 0 \) in the schema. The events (or actions) of the timed automaton of a CSP-OZ specification will always have the form \( Op.i.o \) where \( Op \) is the name of a method and \( i \) and \( o \) (possibly omitted) are values for input and output parameters. This is the CSP view on events. Among others, in our example the separation of the Object-Z-part leads to the following schemas (note that \( reset(note) = \{ x_r, x_f \} \) and \( reset(ring) = \emptyset \)):

\[
\begin{align*}
cl(ring) & \quad \Delta(alarm) \\
\Delta(alarm) & \quad alarm'
\end{align*}
\]

\[
\begin{align*}
cc(note) & \quad \Delta(x_r, x_f) \\
x_r < 10 & \quad x_r' = 0 \land x_f' = 0
\end{align*}
\]

\[
\begin{align*}
cinv(state) & \\
\neg alarm & \Rightarrow x_r \leq 10
\end{align*}
\]

where \( state \) identifies the state schema of the class \( Watchdog \).

The locations of the timed system are (partly) the set of valuations (bindings) of nonclock variables. For such a valuation \( q \) and a state schema \( st \) we write \( q \models st \) if the valuation satisfies the predicates in \( st \); for valuations \( q, q' \), input value \( i \), output value \( o \) and operations schema \( Op \) we write \((q, i, o, q') \models Op \) if \( q \) as before and \( q' \) as after state together with input \( i \) and output \( o \) satisfy the predicate in \( Op \). For understanding the semantics of a timed CSP-OZ class (and for defining timed simulations later) it is useful to think of every class as implicitly having an (infinite) number of operations
Metzler, Wehrheim

Fig. 1. Timed automaton for class Watchdog: Object-Z part

\[ \text{Delay}_d \] for every \( d \in \mathbb{R}_+ \) (a nonnegative real)

\[ \text{Delay}_d \triangleq \left[ \Delta(x_1, \ldots, x_n) \mid x'_i = x_i + d \right] \]

\((x_1, \ldots, x_n)\) the set of all clock variables of the class, advancing the time for \( d \) time units.

The semantics for timed CSP-OZ is now derived in the already mentioned two steps: first, we separately derive a semantics for the part without CSP process equations (called timed Object-Z) and for the CSP part, and in a second step these are combined using the above defined parallel composition operation on timed systems.

**Definition 3.5** Let \( OZ = (\text{State}, \text{Init}, (\text{Op}_i)_{i \in I}) \) be a timed Object-Z class with clock variables \( X = \{x_1, \ldots, x_n\} \) and ordinary variables \( \text{Var} = \{v_1, \ldots, v_m\} \). The semantics of \( OZ, [OZ] \), is the timed system \( T = (Q, \rightarrow, q_0, I) \) with

- \( Q = \{ \rho : \text{Var} \rightarrow D \mid \rho \models \text{cl}(\text{State}), \rho \text{ type correct} \} \) (\( D \) a domain for values),
- \( q \xrightarrow{\text{Op}_i,o,\varphi,Y} q' \) iff \( (q,i,o,q') \models \text{cl}(\text{Op}), \varphi = \text{cc}(\text{Op}) \) and \( Y = \text{reset}(\text{Op}) \),
- \( q_0 \models \text{Init} \),
- \( I(q) = \bigwedge_{\varphi \in \Phi(X)} \exists p : p \Rightarrow \varphi \in \text{cinv}(\text{State}) \land q = p \) \( \varphi \)

Figure 1 shows the timed automaton for the Object-Z part of class Watchdog.

Note that the semantics generates a particular class of timed automata: all transitions labelled with the same event have the same guards and clock conditions. Because of simplicity, we have restricted the clock conditions of a method to be state independent. However, an extension of the semantics to clock conditions relating clock variables and predicates over non-clock variables can be achieved.

The definition of the semantics gives us a close correspondence between the states of the timed Object-Z specification and the configurations of the timed automaton which separate clock valuations from clockless valuations. Given a configuration \( \langle q, \nu \rangle, q : \text{Var} \rightarrow D, \nu : X \rightarrow \mathbb{R}_+ \), we let \( q \oplus \nu : \text{Var} \cup X \rightarrow D \cup \mathbb{R}_+ \) denote the valuation combining the two separate valuations. Then we
get the following relationship between a timed Object-Z specification and its
timed automaton:

**Proposition 3.6** Let $OZ = (\text{State}, \text{Init}, (\text{Op}_i)_{i \in I})$ be a timed Object-Z class and $T = (Q, \rightarrow, q_0, I) = [OZ]$ its semantics. Then the following holds:

(i) $q_0 \oplus \nu_0 \models \text{Init}$.
(ii) $\forall q, \nu, d : (\langle q, \nu \rangle \xrightarrow{d} \langle q, \nu + d \rangle) \iff (q \oplus \nu, q \oplus (\nu + d)) \models \text{Delay}_d$.
(iii) $\forall q, q', \nu, \nu', \text{Op.i.o} : (\langle q, \nu \rangle \xrightarrow{\text{Op.i.o}} \langle q', \nu' \rangle) \iff (q \oplus \nu, i, o, q' \oplus \nu') \models \text{Op}$.

**Proof:**

(i) $q_0 \oplus \nu_0 \models \text{Init}$ follows by the definition of $q_0$ and the fact that we have restricted initialisation of clock variables to 0.

(ii) Implication from left to right:

$\Rightarrow \nu \models I(q), \nu + d \models I(q), \nu \models \text{cl(State)}$ (by definition of the semantics)

$\Rightarrow \nu \models \bigwedge_{\{p \models \varphi \in \text{cmv(State)} \land q = p\}} \varphi$

$\land \nu + d \models \bigwedge_{\{p \models \varphi \in \text{cmv(State)} \land q = p\}} \varphi$

$\land q \models \text{cl(State)}$

$\Rightarrow q \oplus \nu \models \text{State} \land q \oplus (\nu + d) \models \text{State}$

Reverse direction:

$(q \oplus \nu, q \oplus (\nu + d)) \models \text{Delay}_d$.

(iii) Implication from left to right:

$\Rightarrow \exists \varphi, Y : q \xrightarrow{\text{Op.i.o}} q', \nu \models I(q'), \nu \models \varphi, \nu' = \nu[Y := 0], \nu' \models I(q')$

$\Rightarrow \langle q, i, o, q' \rangle \models \text{cl(Op)}, \varphi = \text{cc(Op)}, Y = \text{reset(Op)}, \nu \models \varphi,$

$\nu' = \nu[Y := 0], \nu' \models I(q), \nu \models I(q)$

$\Rightarrow (q \oplus i, o, q' \oplus \nu') \models \text{Op}$.

Reverse direction:

$(q \oplus i, o, q' \oplus \nu') \models \text{Op} \land \varphi = \text{cc(Op)}, Y = \text{reset(Op)}$

$\Rightarrow \nu' = \nu[Y := 0], \nu \models \varphi, \nu \models I(q), \nu' \models I(q'), q \xrightarrow{\text{Op.i.o}, \varphi, Y} q'$

$\Rightarrow \langle q, \nu \rangle \xrightarrow{\text{Op.i.o}} \langle q', \nu' \rangle$. □

The timed system for the CSP part is very simple (no clock conditions at all) and can be derived using the operational semantics of CSP (referred to as $\rightarrow_CSP$ in the next definition) [22]. Note that we do not consider internal events here, thus we restrict ourselves to deterministic CSP processes. This choice is influenced by our notion of implementation which is trace refinement. A distinction between external and internal choice would only be reasonable in the context of a more discriminable semantic model such as the failures-divergences model of CSP, which would thus however have to be extended to
Fig. 2. Timed automaton for class Watchdog: CSP part

the timed setting.

**Definition 3.7** Let `main` be the main process of the CSP part of a timed CSP-OZ class. The semantics of `main`, `[main]`, is the timed system $T = (Q, \rightarrow, q_0, I)$ over an empty set of clock variables where

- $Q$ is the set of CSP terms,
- $q \xrightarrow{Op_i,o,\text{true},\emptyset} q'$ iff $q \xrightarrow{Op_i,o} \text{CSP} q'$,
- $q_0 = \text{main}$ and
- $I(q) = \text{true}$ for all $q \in Q$.

Figure 2 shows the timed automaton for the CSP part of class Watchdog.

The semantics of a timed CSP-OZ specification $C$ consisting of CSP part `main` and Object-Z part $OZ$ is then obtained by combining the semantics of the separate parts using the CSP parallel composition operator on timed systems: $[C] = [OZ] || A [\text{main}]$. The synchronisation set $A$ is the intersection of the alphabets of the CSP part and the Object-Z part. The full timed automaton describing the semantics of class Watchdog is given in Figure 3. The class invariant is now a location invariant (for location $\neg\text{alarm}$), the preconditions of operations referring to clocks are clock conditions on transitions, the predicates $x'_r = 0$ and $x'_f = 0$ become clock resets, and CSP part as well as the clockless part of Object-Z determine the structure of the automaton and the labelling of transitions.

4 Verification

The formal semantics gives us the possibility of analysing interface specifications. Provided the timed automaton has a finite number of locations (which is the case in our example) we can even use a model checker for the analysis.
Several model checkers for timed automata exist; here, we will use UPPAAL [3]. To do so, we first have to describe the timed automaton in the format required by UPPAAL, and then have to formulate (and check) the properties we are interested in.

For the first part, we have to make some global declarations of channels and clocks for UPPAAL. Thus, for our watchdog we declare three channels note, ring and flash corresponding to the methods of the CSP-OZ class. We also define two global clocks $x_r$ and $x_f$ representing the clocks $x_r$ and $x_f$ in our specification.

Next and according to our timed automata of Figure 3, we add a template Watchdog (a skeleton of a timed automaton) to our UPPAAL system. Here, we renamed its states (the names of locations have no influence on the set of timed traces) to facilitate formulation of our verification properties. Each transition is labelled with its corresponding channel name. Finally we add invariants to the states and clock conditions and resets to the transitions as depicted in Figure 3. Figure 4 shows a screenshot of the automaton representing the timed system of our CSP-OZ specification Watchdog.

This now allows us to automatically analyse our interface specification, for instance check whether our three views on the interface (pre/postconditions, protocols and time) in combination give us the desired behaviour. Here, we prove three requirements on our system. The first is deadlock freedom (which

---

6 For simulation and verification purposes, we then later add a second tester component. This component only contains one state and three transitions which synchronize with the three respective channels, i.e. it has no own behavior. This is required since UPPAALs CCS-like synchronisation [19] always expects a partner for a transition labelled with a channel name.
means that it is not possible to find a trace for our timed automaton so that no progress is possible). Deadlock freedom is essential in two different aspects: Considering time constraints, the state invariants and time guards must guarantee that progress is always possible. For example, if we change the invariant on the first ring-transition to \(xr \geq 11\), deadlock freedom would not be ensured: In the open time interval \([10, 11]\), neither any transition is possible nor is it possible to stay in location \(nalarm\_C\). The second aspect – which we do not consider here – is communication between more than one component, where deadlocks based on the CSP part of the classes may occur.

Since the query language of UPPAAL is a subset of CTL \([6]\) (i.e. UPPAAL allows for more than proofs of language inclusion), we show that the formula

\[
A \left[ \not\text{deadlock} \right]
\]

holds. By describing the desired behavior of \(Watchdog\), we want to guarantee that after a certain time, an alarm is raised. That means that we must leave any state representing \(alarm = false\) after at most 10 time units. Therefore we have to verify the formula

\[
A[] \, xr>10 \implies \not (\text{Watchdog.nalarm}M \lor \text{Watchdog.nalarm}C\_1 \lor \text{Watchdog.nalarm}C\_2)
\]

which holds as well. Finally, we do not want the alarm to be raised before 8 time units have passed. We verify that

\[
A[] \, (\text{Watchdog.alarm}R \lor \text{Watchdog.alarm}F) \implies xf>=8
\]

holds.
5 Timed simulations and implementation

Next, we are interested in showing language inclusion, i.e. an implementation relationship between a higher level timed CSP-OZ specification \( A \) and its implementation \( C \). Language inclusion is the correctness criterion that we intend to use for substitutability and interoperability checks of component interfaces. For instance, if we are given two component specifications \( A \) and \( C \) defining what the components require from other components, \( A \) can be safely replaced by \( C \) if the language of \( C \) is a subset of that of \( A \) (it requires less).

Language inclusion for timed automata is in general undecidable [1]. Here, we give an approach to checking language inclusion of timed CSP-OZ specifications. The additional structure present in the specifications allow for a compositional language inclusion check, separately treating the CSP and the Object-Z part. The check for the Object-Z part has to be carried out manually, the check on the CSP part can be done by a CSP model checker.

We start with developing timed simulation conditions for timed Object-Z specifications. These can be used to carry out an operation-wise proof of language inclusion between two timed Object-Z specifications. The conditions can be seen as one half of timed bisimulations [26], and are similar to those in approaches specifying components by pre- and postconditions of their operations. For showing a simulation relation between an abstract and a concrete Object-Z specification we have to give a relation \( R \) relating the state spaces of both specifications:

**Definition 5.1**

Let \( A = (AInit, AState, (AOp_i)_{i \in I}) \) and \( C = (CInit, CState, (COp_i)_{i \in I}) \) be two timed Object-Z specifications. \( C \) is a timed simulation of \( A \) if there is a relation \( R : CState \leftrightarrow AState \) such that the following hold:

1. **(I)** \( \forall CState \cdot CInit \Rightarrow \exists AState \cdot AInit \land R \)
2. **(C1)** \( \forall CState, CState', AState \cdot R \land COp \Rightarrow \exists AState' \cdot AOp \land R' \)
3. **(C2)** \( \forall CState, CState', AState, d \cdot R \land Delay_d^C \Rightarrow \exists AState' \cdot Delay_d^A \land R' \)

Condition (I) is an initialisation condition requiring every initial state of \( C \) to have a corresponding initial state in \( A \). Conditions (C1) and (C2) are correctness conditions which state that both delay and action steps of \( C \) have corresponding steps in \( A \). Note that unlike downward or backward simulation conditions for data refinement we do not have separate applicability conditions here. This is justified by the notion of implementation for timed automata which is simply language inclusion. Language inclusion does not require availability of methods to carry over from the abstract to the concrete system. Thus correctness is sufficient here. Note that the rules are sound but not complete for language inclusion.
Theorem 5.2
Let \( A = (A_{\text{Init}}, A_{\text{State}}, (A_{Opt})_{i \in I}) \) and \( C = (C_{\text{Init}}, C_{\text{State}}, (C_{Opt})_{i \in I}) \) be two timed Object-Z specifications. Then, the following holds:

If \( C \) is a timed simulation of \( A \) then \( \mathcal{L}(\llbracket C \rrbracket) \subseteq \mathcal{L}(\llbracket A \rrbracket) \).

Proof: The proof relies on Proposition 3.6 showing the correspondence between the states of a timed Object-Z specification and the configurations of its timed automaton.

Assume \((\sigma, \tau) \in \mathcal{L}(\llbracket C \rrbracket)\). Then there is an execution of \( \llbracket C \rrbracket \) over \((\sigma, \tau)\), i.e.

\[
\langle q^C_0, \nu^C_0 \rangle \xrightarrow{d_i, a_i} \langle q^C_1, \nu^C_1 \rangle \xrightarrow{d_2, a_2} \langle q^C_2, \nu^C_2 \rangle \ldots
\]

such that \( \tau_i = \tau_{i-1} + d_i \) for all \( i > 1 \). We inductively construct an execution of \( \llbracket A \rrbracket \) over \((\sigma, \tau)\).

Induction base: By Proposition 3.6 we get \((q^C_0 \oplus \nu^C_0) \models C_{\text{Init}}\). Hence, by (I) there is a state \( q^A_0 \) such that \((q^A_0 \oplus \nu^A_0) \models A_{\text{Init}}\) and \((q^C_0 \oplus \nu^C_0, q^A_0 \oplus \nu^A_0) \in R\).

Induction hypothesis: We assume to have constructed an execution sequence of \( A \) up to some index \( i \)

\[
\langle q^A_0, \nu^A_0 \rangle \xrightarrow{d_i, a_i} \langle q^A_1, \nu^A_1 \rangle \xrightarrow{d_2, a_2} \ldots \langle q^A_i, \nu^A_i \rangle
\]

such that \( \forall j \leq i : (q^C_j \oplus \nu^C_j, q^A_j \oplus \nu^A_j) \in R \).

Induction step: Assume \((q^C_i \oplus \nu^C_i, q^A_i \oplus \nu^A_i) \in R\). If \( \langle q^C_i, \nu^C_i \rangle \xrightarrow{a_i} \langle q^C_i, \nu^C_i + d \rangle \) then by Proposition 3.6 \((q^C_i \oplus \nu^C_i, q^C_i \oplus (\nu^C_i + d)) \models \text{Delay}_d^C\). By (C2) it follows that there is some state \( s^A \) such that \((q^A_i \oplus \nu^A_i, s^A) \models \text{Delay}_d^A\) and \((q^C_i \oplus (\nu^C_i + d), s^A) \in R\). By definition of the Delay operation it follows that \( s^A = q^A_i \oplus (\nu^A_i + d) \). Again by Proposition 3.6 we thus get \( \langle q^A_i, \nu^A_i \rangle \xrightarrow{a_i} \langle q^A_i, \nu^A_i + d \rangle \). The case of action transitions can be shown in an analogue way.

The fact that this is an execution of \( A \) over \((\sigma, \tau)\) follows since the same \( d_i \) and \( a_i \) appear in the execution of \( A \) (and hence the \( \tau_i \) will be the same).

We will exemplify timed simulations by the following implementation \texttt{ImpWatchdog} of class \texttt{Watchdog} which resolves some of the nondeterminism and uses one clock only. The timed language of \texttt{ImpWatchdog} is a subset of that of \texttt{Watchdog}.

<table>
<thead>
<tr>
<th>\texttt{ImpWatchdog}</th>
</tr>
</thead>
<tbody>
<tr>
<td>method \texttt{flash}</td>
</tr>
<tr>
<td>method \texttt{note}</td>
</tr>
<tr>
<td>\texttt{main = note \rightarrow main \ □ flash \rightarrow Alarm}</td>
</tr>
<tr>
<td>\texttt{Alarm = flash \rightarrow Alarm}</td>
</tr>
</tbody>
</table>
A timed simulation between ImpWatchdog and Watchdog can be shown with the following representation relation between states of ImpWatchdog and Watchdog:

The proof involves a number of steps:

(I) Initialisation amounts to proving

\[ \forall \text{warning}, x \cdot \neg \text{warning} \Rightarrow \exists \text{alarm}, x_r, x_f \cdot \neg \text{alarm} \land x_r = 0 \land x_f = 0 \land R \]

which holds (use \( \neg \text{alarm} \land x_r = 0 \land x_f = 0 \)).

(C1) Correctness for operation note is

\[ \forall \text{warning}, x, \text{warning}', x', \text{alarm}, x_r, x_f \cdot R \land \neg \text{warning} \land x < 8 \land x' = 0 \Rightarrow \exists \text{alarm}', x_r', x_f' \cdot \neg \text{alarm} \land x_r < 10 \land x_r' = 0 \land x_f' = 0 \land R' \]

which holds by using \( \neg \text{alarm}' \land x_r' = 0 \land x_f' = 0 \).

Correctness for operation flash is

\[ \forall \text{warning}, x, \text{warning}', x', \text{alarm}, x_r, x_f \cdot R \land x \geq 8 \land \text{warning}' \Rightarrow \exists \text{alarm}', x_r', x_f' \cdot x_f' \geq 8 \land \text{alarm}' \land R' \]

where \( \text{alarm}' \land x_f' = x' \land x_r' = x' \) is a solution.

(C2) Although there are an infinite number of operations Delay\(_d\) (one for

\[ \begin{array}{c|c}
\text{warning} & \text{Init} \\
\text{x : Clock} & \neg \text{warning} \\
\neg \text{warning} \Rightarrow x \leq 8 & x = 0 \\
\end{array} \]

\[ \begin{array}{c|c}
\text{note} & \text{flash} \\
\Delta(x) & \Delta(\text{warning}) \\
\neg \text{warning} & x \geq 8 \\
x < 8 \land x' = 0 & \text{warning}' \\
\end{array} \]
Metzler, Wehrheim

each $d$), there is just one proof to be done, namely

$$
\forall \text{warning}, x, \text{warning}', x', \text{alarm}, x_r, x_f, d \bullet
R \land \text{warning}' = \text{warning} \land x' = x + d
\Rightarrow \exists \text{alarm}', x_r', x_f' \bullet \text{alarm}' = \text{alarm} \land x_r' = x_r + d \land x_f' = x_f + d \land R'
$$

which holds immediate, since clock values are all equal. Thus class

$\text{ImpWatchdog}$ is indeed an implementation of class $\text{Watchdog}$.

This so far only gives us a means for showing an implementation relationship between pure timed Object-Z specifications. In addition, we have to look at the CSP parts and have to check whether an implementation relationship between the timed automata of the processes $\text{main}_A$ and $\text{main}_C$ (where $\text{main}_A$ is the CSP process of the abstract class $A$ and $\text{main}_C$ that of the more concrete class $C$) holds as well. Here, the check is very easy: it amounts to checking trace refinement between the CSP processes. Trace refinement is one of the notions of refinement supported by the semantic model of CSP, and – given the CSP processes are finite state – can be checked using the CSP model checker FDR [10]. Trace refinement is just language inclusion (again assuming no acceptance states) in the timeless setting. Since the timed automata for the CSP processes pose no restrictions on time, trace refinement is sufficient:

**Lemma 5.3** Let $\text{main}_A$ and $\text{main}_C$ be the CSP processes of timed CSP-OZ classes $A$ and $C$. If $\text{main}_C$ is a trace refinement of $\text{main}_A$ then $\mathcal{L}(\text{main}_C) \subseteq \mathcal{L}(\text{main}_A)$.

Finally, these two results have to be integrated into one. So far, we have some means for showing an implementation relationship for the timed Object-Z parts and for the CSP parts. These techniques can separately be applied to specifications if the implementation relationship is preserved under the operator that we use for combining the semantics of the separate parts, namely under parallel composition. The following theorem states exactly this property.

**Theorem 5.4** Let $S_i = (P_i, \rightarrow^S_i, p_{0,i}, I_i)$, $i = 1, 2$, be timed systems over $\Sigma_1$ and $T_i = (Q_i, \rightarrow^T_i, q_{0,i}, J_i)$, $i = 1, 2$, over $\Sigma_2$ with a disjoint set of clock variables $X_i$ for $S_i$ and $Z_i$ for $T_i$. Let $A = \Sigma_1 \cap \Sigma_2$ be the synchronisation set. Then the following holds:

$$
\mathcal{L}(S_1) \subseteq \mathcal{L}(S_2) \land \mathcal{L}(T_1) \subseteq \mathcal{L}(T_2) \Rightarrow \mathcal{L}(T_1 \parallel_A S_1) \subseteq \mathcal{L}(T_2 \parallel_A S_2)
$$

The proof of this theorem relies on a certain property of delay transitions (that itself depends on the form of clock conditions for invariants) which allows us to combine and decompose delay transitions. Here, we just state and give the proof of this property.

**Lemma 5.5** Let $T = (Q, \rightarrow, q_0, I)$ be a timed system. Then
∀ d, d_1, d_2 \bullet d = d_1 + d_2 \bullet
\langle q, \nu \rangle \xrightarrow{d_1} \langle q, \nu + d_1 \rangle \iff \langle q, \nu + d_1 \rangle \xrightarrow{d_2} \langle q, \nu + d_1 + d_2 \rangle

Proof: The direction \iff follows immediately since by definition one can assume that the intermediate configuration \langle q, \nu + d_1 \rangle can be skipped.

Let \langle q, \nu \rangle \xrightarrow{d_1} \langle q, \nu + d_1 \rangle and \quad d = d_1 + d_2 with d_1, d_2 \in \mathbb{R}_+ \cdot By assumption we know that \nu \models I(q) and \nu + d_1 + d_2 \models I(q). We need to prove that
\langle q, \nu \rangle \xrightarrow{d_2} \langle q, \nu + d_1 \rangle \xrightarrow{d_2} \langle q, \nu + d_1 + d_2 \rangle,

meaning \nu + d_1 \models I(q). We do this by induction on the \varphi \in \Phi(X):

Induction base:
• \varphi = (x = c):
By \nu \models x = c and \nu + d_1 + d_2 \models x = c we get d_1 = d_2 = 0, \text{i.e. } \nu + d_1 \models x = c, since we do not reset the time in between our delays.

• \varphi = (x \leq c):
We have \nu \models x \leq c and \nu + d_1 + d_2 \models x \leq c. Again we immediately deduce \nu + d_1 \models x \leq c

• \varphi = (x \geq c), \varphi = (x > c), \varphi = (x < c):
The same arguments are used to show these properties.

Induction step: Let \varphi = \psi_1 \land \psi_2. Since \nu \models \varphi and \nu + d_1 + d_2 \models \varphi, it follows that \nu \models \psi_1, \nu \models \psi_2, \nu + d_1 + d_2 \models \psi_1 and \nu + d_1 + d_2 \models \psi_2. By induction hypothesis we deduce that \nu + d_1 \models \psi_1 and \nu + d_1 \models \psi_2, i.e. \nu + d_1 \models \psi_1 \land \psi_2.

Finally, we can combine Theorem 5.2 and Lemma 5.3 in the following way:

Corollary 5.6 Let main_A and main_C be the CSP processes of timed CSP-OZ classes A and C, let OZ_A and OZ_C be their timed Object-Z-parts. If main_C is a trace refinement of main_A and OZ_C is a timed simulation of OZ_A, then \mathcal{L}([C]) \subseteq \mathcal{L}([A]), \text{i.e. } C \text{ is an implementation of } A.

We thus can separately check for language inclusion.

6 Conclusion

In this paper we have proposed an extension of CSP-OZ with features for specifying timing constraints on components. The extension has been minimal in the sense that we neither added a third formalism to CSP-OZ nor exchanged one of the notations which have already been integrated into CSP-OZ. The only extension is a new type for variables in Object-Z. We believe this to be an extension which makes reading and writing of specification particularly easy: once a designer is familiar with CSP and Object-Z he/she can easily read and write timed CSP-OZ specifications.
For this new formalism we gave a formal semantics, showed, by means of an example, how existing model checkers can in principle be used to analyse components interfaces and discussed simulation conditions for implementation relationships. This now gives us a high-level specification language for components which offers a richer set of facilities for modelling than timed automata do. The additional structure present in timed CSP-OZ specifications could furthermore facilitate static analysis of specifications which might prove fruitful for verification. By restricting our semantics to the traces model, we obtain a simple definition of timed simulations which is very close to the techniques and definitions used in the context of timed automata. Nevertheless we want to deal with parallel composition of several components which calls for a more precise semantics. Future work sees the expansion of our approach to the failures-divergences model of CSP [23] and the use of parallel composition and nondeterminism.

**Related work.**

The combination of an already existing formalism with time is subject of intense research and most often used in the context of safety-critical component-based systems. For example, [15] use OCL for specifying contracts with time and also use an underlying semantics over timed automata.

There are a number of existing integrated notations like CSP-OZ which allow for the description of timing requirements. The approach most often chosen is that of a combination with timed CSP [23]: [17] and [7] combine Object-Z with timed CSP and [25] combines Z with timed CSP. The main difference to these combinations is our semantic model of timed automata. This immediately allows us to use existing standard model checkers for timed systems (when our specifications are finite state).

The use of timed automata (in combination with timed MSCs) for describing components has also been proposed in [5]. Timed automata are therein used for describing individual components and timed MSCs the interaction between components. The two formalisms are, however, not integrated so far; there is no semantics describing the meaning of this combined use of timed automata and MSCs. An approach which uses timed automata like structures to define *timed connectors* of components can be found in [2].

The two approaches most closest to us are CSP-OZ-DC [14] and a recent combination of Object-Z with timed automata [8]. The former is an extension of CSP-OZ with Duration Calculus, which is an interval logic for describing timing requirements. The main difference to our work can be seen in the style of specification: while Duration Calculus allows for a declarative formulation of timing constraints, our approach takes a more operational style. The semantics of CSP-OZ-DC is formulated in terms of phase-event-automata in which clocks then explicitly appear. Model checking on this type of automata can be carried out using a constraint-solving based checker [13]. Since our timed automata are close to phase-event-automata a similar type of verification could
be possible for timed CSP-OZ.

The combination of Object-Z and timed automata proposed in [8] also adds clock variables to Object-Z. These variables may however only be used in the timed automaton which in addition may appear in an Object-Z class. Refinement or implementation relationships are so far not treated.

References


