HAL
open science

A process algebraic framework for specification and validation of real-time systems

Adnan Sherif, Ana Cavalcanti, He Jifeng, Augusto Sampaio

To cite this version:

Adnan Sherif, Ana Cavalcanti, He Jifeng, Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. Formal Aspects of Computing, 2009, 22 (2), pp.153-191. 10.1007/s00165-009-0119-6 . hal-00534927

HAL Id: hal-00534927

https://hal.science/hal-00534927v1

Submitted on 11 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOI 10.1007/s00165-009-0119-6
BCS © 2009
Formal Aspects of Computing (2010) 22: 153–191

Formal Aspects of Computing

A process algebraic framework for specification and validation of real-time systems

Adnan Sherif[1], Ana Cavalcanti[2], He Jifeng[3] and Augusto Sampaio[1]

[1] Centro de Informática, Universidade Federal de Pernambuco, Recife, PE, Brazil
[2] Department of Computer Science, University of York, York, UK. E-mail: Ana.Cavalcanti@cs.york.uk
[3] Software Engineering Institute, East China Normal University, Shanghai, China

Abstract. Following the trend to combine techniques to cover several facets of the development of modern systems, an integration of Z and CSP, called *Circus*, has been proposed as a refinement language; its relational model, based on the unifying theories of programming (UTP), justifies refinement in the context of both Z and CSP. In this paper, we introduce *Circus Time*, a timed extension of *Circus*, and present a new UTP time theory, which we use to give semantics to *Circus Time* and to validate some of its laws. In addition, we provide a framework for validation of timed programs based on FDR, the CSP model-checker. In this technique, a syntactic transformation strategy is used to split a timed program into two parallel components: an untimed program that uses timer events, and a collection of timers. We show that, with the timer events, it is possible to reason about time properties in the untimed language, and so, using FDR. Soundness is established using a Galois connection between the untimed UTP theory of *Circus* (and CSP) and our time theory.

Keywords: Relational models; Unifying theories of programming; CSP; Galois connections

1. Introduction

First generation real-time applications were relatively simple and did not involve sophisticated algorithms or extensive computational complexity. In the last decades, however, there has been a demand for more complex and safety-critical applications, in areas such as aerospace navigation and control, factories monitoring, and nuclear power plants. For these modern systems, the choice of specification language is an important factor in the success of the entire development. The language should cover several facets of the requirements, and should have a model suitable to study the behavior of the system and to establish the validity of desired properties. Such a formal model can also be the basis for a refinement into executable code.

During the last few years, researchers have developed a large number of formal specification languages. Many of them are suitable for expressing and analyzing particular characteristics of a system. For example,

Correspondence and offprint requests to: A. Cavalcanti, E-mail: Ana.Cavalcanti@cs.york.ac.uk

Z [Toy02, WD96] is a formal language used to define data types and to show the effect of operations on these types. It lacks, however, features to express the order in which the operations are executed [Eva94]. Process algebras, like CSP [Hoa85, Ros98], on other hand, are suitable for showing the order of the occurrence of events but lack the ability to handle complex abstract data types and operations. Finally, formalisms like temporal logics and its derivatives [Pnu77, BH81] concentrate on time aspects.

The increasing complexity of real-time systems has made the use of formal specification more frequent in this area, and new languages have proliferated. Timed CSP [RR88] is an extension of CSP that includes new operators like *Wait* and $\triangleright$ (timeout). It has a new semantic model, derived from the untimed models of CSP, to represent dense time information, and a proof system [DS95, Sch00]. Timed CCS [Che93] has an operational semantics; a communication on a described by $a(t)_e^{e'}$ is restricted to happen in the closed time interval $[e, e']$, and, after the communication, the variable t holds the time at which it occurred. Proof rules for timed CCS have been elaborated and shown to be independent of the time domain used. A detailed comparison of several timed process algebras can be found in [EHLW97].

Logic approaches to specification benefit from clear notations and automated validation using existing theorem provers. Modal logic [Che99] adds time reasoning in logical formulae. Temporal logic is a modal logic in which new operators for quantification either in the future or in the past are included. To overcome difficulties with modularity, temporal logic is often used in combination with other techniques. An example is the work of Duke and Smith [DS89], in which temporal logic is used in the invariants of Z specifications to define liveness properties. Lamport in [Lam94], on other hand, adds the concept of actions to classical temporal logic. Duration calculus (DC) [CHR91] is concerned with intervals instead of time instances. Real-time logic (RTL) [JM86] extends predicate logic by relating events with the time in which they occur. In contrast to other logics, RTL allows specification of the absolute timing of events, and not only their relative ordering; it also provides a uniform way of incorporating different scheduling disciplines.

Timed automata [AD94] extend state-transition systems with finitely many real-valued clock variables that are used in annotations. Analysis is based on a finite quotient of the infinite space of clock valuations. Work has been carried out on verification algorithms, including heuristics, and tools [BLL+95, Bey01].

Petri Nets allow mathematical modeling of discrete event systems in terms of conditions and events, and the relationship between them [Mur89]. Time information has been added to Petri Nets in a number of forms. The most common approach is to add time delays to transitions. A similar approach assigns delays to places instead of transitions, and creates a delay between the time the token arrives in a place and the time it enables a transition to fire. A more flexible approach assigns intervals to the transitions and time stamps to the tokens. Such an approach is used by Time Basic Nets (TBNet) [GMMP89].

Specifications of complex systems normally involve a mixture of data types, operations, and time constraints; current research has focused on more comprehensive languages. *Circus* [WC02] combines CSP, Z, specification statements [Mor94] and guarded commands [Dij76] to provide a notation for both specification and programming, and for verification by refinement. Laws are explored in [CSW03] and tools are under development [WCF05, FC06]. Several combinations of a process algebra with a state-based formalism have been proposed [Fis98, TS99, MD00]; the distinguishing features of *Circus* are its refinement theory and technique.

The combination of different formalisms in *Circus*, as well as its refinement theory, is justified in the semantic framework of the UTP [HH98, OCW07a]. It proposes the use of relations as a basis for unification of different programming paradigms, and studies their relationship using mappings that associate programs in different relational theories. Here, we propose a new UTP theory to capture discrete time information; it extends the existing *Circus* theory [OCW07b]. The new model can be used to specify and reason about hard and soft real-time systems; it is also suitable for representing periodic and aperiodic tasks.

Time is continuous by nature, but a discrete representation of time is also satisfactory in most cases. In specification languages, time is often represented by real numbers, but in programming languages, time is represented by integers. In principle, the continuous time model is more appropriate because it can express time in both forms, and time in the real world is continuous. A continuous time model, however, cannot be implemented by a software system. Since we aim at a language for refinement, which can be used for programming as well as specification, we adopt a discrete model. As a consequence, the original untimed refinement laws of *Circus* can be extended in a natural way.

Works like those in [LH99, HV02], which use Extended Duration Calculus (EDC) to add continuous time to language semantics, show the elegance and powerful expression capacity of the EDC formulas. Both approaches, however, make it clear that the new model cannot be easily related to the original untimed model. Proving properties in the new model is a tedious task.

The addition of time operators to *Circus* was first discussed in [SH02], where we proposed a new refinement language, *Circus Time*. We considered the impact of the extra operators on the *Circus* UTP model, but we did not present a complete theory. We omitted, for example, a definition of parallelism, and did not study healthiness conditions or any other algebraic properties of the language or UTP theory.

In [SHCS05] we have also presented an initial proposal for a validation framework. It is based on the reduction of a timed program to a normal form that partitions it into an untimed program that uses timer events, and a set of timers. The untimed program relies on timer events to preserve the semantics of the original timed program. In this way, the framework allows reasoning in the untimed model to establish properties of timed programs. Its soundness, however, was not formally addressed in [SHCS05].

In this paper, we both completely formalize the *Circus Time* model, and establish the soundness of our reasoning framework. Our new UTP model is a semantic characterization of the *Circus Time* operators. The result is a detailed description of a UTP theory for state-rich timed reactive systems; following the style of the UTP, we define the relevant observational variables, the healthiness conditions that characterise feasibility, and programming operators. An extensive set of algebraic laws clarifies how the operators interact. In addition, we formalise the relationship between our new theory and the existing UTP *Circus* theory for untimed state-rich reactive systems. We conclude that the original model is an abstraction of the time model, and that there is a Galois connection between them. The new operators of *Circus Time* are not significantly different from those of Timed CSP, for example, but by considering them in the context of the UTP, we make a contribution to unification and integration. *Circus Time* is an example of how these results can be used in the context of a language much richer than CSP. The UTP, of course, provides a foundation for many other paradigms, which can be combined and related in a uniform framework.

We also provide here the semantics of new operators required by the reasoning framework, namely, variations of external choice and parallel composition, and show that their algebraic characterisation is complete by providing a reduction strategy that eliminates them. To establish the soundness of the validation strategy, we prove that when an untimed program generated during normalisation and its timers are composed in parallel, the resulting program is equivalent to the original one: normalisation preserves the semantics of timed programs. In addition, we prove that if the timers of the normalised specification and those of the normalised implementation are the same, then if refinement holds for the untimed programs in the untimed model, then it holds for the timed programs as well.

This justifies the use of untimed tools, like the CSP model-checker FDR [For97], to analyse timed specifications. In this approach, the specification of requirements and the implementation are both described using *Circus Time*, and normalised by applying a syntactic transformation. We can then use FDR to show that the program meets its specification, dealing only with the untimed components of each normal form.

The approach that we follow is based on well established work on Timed CSP, which has been applied in a variety of areas: avionics, protocols, and robotics are a few examples. Our main contributions are a semantic model that can be used for state-rich languages, of which *Circus Time* is an example, and its relationship to the existing untimed model, as explored in the reasoning framework. These results have already inspired others to model different languages using (extensions of) our UTP theory [SH03, QDC03, BSW07].

In the next section, we present *Circus Time*. It is a subset of *Circus*, extended with two new constructs related to time constraints. An extension of the *Circus* UTP theory is presented, and used to defined the semantics of *Circus Time*. In Sect. 3 we explore the relationship between the timed model of *Circus Time* and the untimed model of *Circus*; mapping functions are used to relate the two models. Our validation framework is described in Sect. 4. Finally, in Sect. 5, we present our conclusions, and discuss related and future work. Appendix A lists the definitions of *Circus Time* operators that are very similar to those of the corresponding CSP operators, and therefore do not require detailed discussion, but may be used in some proofs, and Appendix B gives a few laws of *Circus Time* used in the proofs.

2. *Circus Time* and its model

This section introduces *Circus Time*: we describe the language informally, and provide a brief introduction to the UTP, before presenting our UTP theory for *Circus Time*.

$$
\begin{array}{rcl}
\mathsf{Action} & ::= & \mathit{Skip} \mid \mathit{Stop} \\
 & \mid & \mathit{Wait}\ \mathsf{t} \mid \mathit{Chaos} \\
 & \mid & \mathsf{Communication} \rightarrow \mathsf{Action} \mid \mathsf{b}\ \&\ \mathsf{Action} \\
 & \mid & \mathsf{Action} \sqcap \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \\
 & \mid & \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \llbracket \mathsf{NS} \mid \mathsf{CS} \mid \mathsf{NS} \rrbracket \mathsf{Action} \\
 & \mid & \mathsf{Action} \setminus \mathsf{CS} \mid \mathsf{Action} \overset{\mathsf{t}}{\rhd} \mathsf{Action} \\
 & \mid & \mathsf{N} := \mathsf{e} \mid \mathsf{Action} \lhd \mathsf{b} \rhd \mathsf{Action} \\
 & \mid & \mu\, \mathsf{N} \bullet \mathsf{Action} \mid \mathsf{N} \\
\mathsf{Communication} & ::= & \mathsf{N}\ \mathsf{CParameter}^* \\
\mathsf{CParameter} & ::= & ?\,\mathsf{N} \mid !\,\mathsf{e} \mid .\,\mathsf{e}
\end{array}
$$

Fig. 1. *Circus Time* syntax

2.1. *Circus Time*: syntax and informal description

A *Circus* program defines a collection of processes that encapsulate a state defined using Z, and exhibits a behaviour defined by a main action using a mixture of Z data operations, specification statements, assignments and conditionals, and CSP constructs. Processes can also be combined using CSP constructs.

For our study of *Circus Time*, we concentrate on the notation for definition of actions, but observe that processes can be defined in *Circus Time* just as they are defined in *Circus*, since our *Circus Time* theory is also rich enough to cope with states. To simplify its presentation, though, we assume here the existence of a global state, and allow assignments to introduce new variables. We also omit specification constructs, which again have definitions similar to those in *Circus*, in as much as the definition of assignment in the *Circus Time* theory is similar to that in the original *Circus* theory.

The challenges for our new model and technique arise from the free combination of data operations, communications, choices and parallelism, and timed constructs; recursion also imposes important restrictions. All these constructs are included in *Circus Time*; Fig. 1 presents a BNF description of the syntax. In this figure, b stands for a predicate, e for any expression, t for a positive integer expression, N for any valid name (identifier), NS for a set of variable names, and CS for a set of channel names.

A *Circus Time* program is formed of one single action. An action can be basic or a combination of one or more actions. *Skip* is a basic action that terminates immediately. *Stop* represents deadlock; it is a program in an ever waiting state. *Wait* t puts the program in a waiting state for a period of time determined by the positive integer t. *Chaos* is the worst action; nothing can be guaranteed about its behaviour.

An action can be prefixed with a communication, which takes place before the action starts. A communication can be a simple synchronisation c over a channel c, an input communication $c?x$, which assigns to x the value input through the channel c, or an output communication $c!e$ (or $c.e$), which outputs the value of the expression e through c. A communication can also involve a combination of inputs and outputs; in Fig. 1, CParameter* is a list of zero or more communication parameters defining inputs and outputs. For an arbitrary communication c, the prefixed action $c \rightarrow A$ waits for the other actions that need to synchronise on c before the communication can take place, and then it behaves like A. If the communication is an input, the scope of the input variable is restricted to A. In $b\ \&\ A$, b is a guard: a boolean expression that has to hold for the action A to take place; otherwise we have a deadlock.

The internal choice $A \sqcap B$ arbitrarily selects A or B for execution. The external choice $A \Box B$ waits for interaction with the environment; the first action that engages on a communication or terminates is chosen. The sequential composition $A;\ B$ behaves as A followed immediately by B. The parallel composition $A \llbracket s_A \mid cs \mid s_B \rrbracket B$ of actions A and B determines the set cs of channels through which communication requires synchronisation, and disjoint sets s_A and s_B of variables that they can change. The variables that do not appear in the set associated with the action cannot be changed by it; however, all variables can be accessed by both A and B, and their values are those that they hold before the parallelism starts. In an action $A \setminus cs$ the communications through the channels in the set cs are hidden, that is, internal to A; hidden channels cannot be used for interaction with other actions.

The timeout action $A \overset{t}{\rhd} B$ is a time guarded choice; its behaviour is that of either A or B. If A performs an observable event or terminates before the specified time t elapses, it is chosen. Otherwise, A is suspended and the only possible observations are those produced by B.

The assignment action simply assigns a list of values to a corresponding list of variables in the current state. If the variable already exists, its value is overwritten; otherwise it is added to the current state. The conditional action $A \lhd b \rhd B$ executes A if b evaluates to *true*, otherwise the action B is executed. Finally, $\mu X \bullet F(X)$ defines a recursive action F; any reference to X within it stands for a recursive call.

As an example, we present a specification of an alarm system [TM87]. It is a common burglar alarm controller connected to sensors which detect movements or changes in the environment. When disabled, the controller ignores any disturbance; when enabled, it will sound an alarm when a sensor signals a disturbance. There are two timing requirements on the alarm controller. Firstly, after enabling the alarm controller, there is a period T_1 before a disturbance is detected. This period permits a person to enable the alarm and get out. Secondly, when a disturbance is detected, the controller will wait for a period T_2 before activating the alarm. This will allow a person to enter the building and deactivate the alarm.

We use the event *enable*, that is, a synchronisation on the channel *enable*, to indicate that the alarm system is enabled. To disable the alarm, the event *disable* is used. When the alarm system is disabled it responds only to the event *enable*. The event *disturbed* indicates that a sensor has detected a disturbance. Finally, *alarm* signals the firing of the alarm. For clarity of presentation, we name a few actions, *Disable*, *Running*, and *Active*, which are composed in the description of the action *Alarm* that defines our system.

$$Disable = disable \rightarrow Skip$$
$$Running = Disable \Box (disturbed \rightarrow Active)$$
$$Active = Disable \overset{T_2}{\rhd} (alarm \rightarrow Disable)$$
$$Alarm = \mu X \bullet (enable \rightarrow (Disable \overset{T_1}{\rhd} Running));\ X$$

The action *Disable* simply offers to engage on the event *disable*. The action *Running* represents the armed behaviour of the alarm controller: the controller can either be disabled or it can be *disturbed*. When the alarm is disturbed, it behaves as *Active*, which models the active state of the alarm. In this state, the controller can again be disabled for the first T_2 time units; after this period an *alarm* is fired. When fired, the controller will only terminate once disabled. The main action *Alarm* is recursive. The controller starts by assuming that the alarm is disabled and offers the *enable* event to start the controller. After the event *enable*, the action can be disabled for the first T_1 time units before it is armed.

In Sect. 2.3, we give a formal UTP definition of the *Circus Time* constructs.

2.2. Unifying theories of programming

In the unifying theories of programming of Hoare and He [HH98], specifications and programs are all called programs, and interpreted as relations (defined as predicates) between an initial observation and a single subsequent (intermediate or final) observation of the behaviour of a device executing a program. Programming paradigms are differentiated by their *alphabet*, *signature*, and a selection of laws known as *healthiness conditions*; together, they characterise a UTP theory. The *alphabet* of a theory gives names for external observations of program behaviour. Like in the Z notation, the UTP uses the convention that the name of an initial observation is undecorated, but the name of a similar observation taken subsequently is decorated with a dash. The set of undecorated names of the alphabet is called the input alphabet, and the set of dashed names is the output alphabet. The signature of a theory provides syntax for denoting the predicates (and expressions) of the theory. The *healthiness conditions* identify the valid predicates.

The *alphabet* of a theory is determined by the observations relevant to understand programs in the particular paradigm of interest. For example, in the theory of reactive programs, a boolean observation variable *wait* distinguishes an intermediate observation from one of a terminated program. In the theory of communicating processes, the variable *tr* is a sequence of events that records the interactions between the process and its environment, and *ref* is a set of interactions that can be refused by the process. In both of these theories, the boolean variable *ok* records whether the program has been properly started in a stable state, and ok' records subsequent stabilization in an observable state. This permits a description of divergent programs: a stable state is either a termination state or a state in which the program is waiting for an interaction with its environment; if a program diverges, then it does not reach a stable state.

The alphabet of the UTP theory for CSP includes *ok*, *wait*, *tr*, and *ref*, and their dashed counterparts. This theory, however, does not preclude the existence of extra observational variables corresponding to programming variables. In this sense, and in a few others, it is already richer than the failures–divergences model of CSP. Indeed,

in the *Circus* theory the programming variables, and their dashed counterpart are in the alphabet, and in the *Circus Time* theory presented in Sect. 2.3, we have observational variables *state* and *state*′ to record the initial and subsequent values of the variables in scope.

A healthiness condition identifies feasible descriptions. Typically, there are healthiness conditions associated with each observation variable in the alphabet, and with groups of related variables. For example, if a program p has not started, observations of its behaviour are impossible, and no predictions can be made. This is captured by a healthiness condition that requires programs p to satisfy the following equation.

$$p = ok \Rightarrow p$$

If ok is *true*, then $ok \Rightarrow p$ is p itself: p started and its behaviour is described by p. On other hand, if ok is *false*, then $ok \Rightarrow p$ is *true*, which means that the behaviour of p is not restricted.

Alternatively, we can define this healthiness condition in terms of a function on predicates $\mathsf{H}(p) = ok \Rightarrow p$; the healthy programs, or predicates, are the fixed points of H. We use the term healthiness condition to refer both to these functions and to the requirement that predicates are one of their fixed points. All healthiness conditions are idempotent and monotonic with respect to refinement. They can be applied to an unhealthy predicate to make it healthy, and so have an important role in linking theories.

The most general UTP theory includes only programming variables (and their dashed counterparts) in the alphabet; there are no healthiness conditions. In this theory, we already have definitions for programming operators like sequence, conditional, and others. Sequence corresponds to relational composition. Provided the output alphabet of p is the same as the input alphabet of q, except, of course, for the decorations, then we can define the sequence $p;\ q$ as shown below, where v is a list of the variables in the input alphabet of q.

$$p;\ q \mathrel{\widehat{=}} \exists\, v_0 \bullet p[v_0/v'] \wedge q[v_0/v]$$

The list of variables v_0 is obtained by 0-subscripting those in v; they represent their intermediate values. The substitutions replace the dashed variables of p and the undashed variables of q with the corresponding 0-subscripted variables, which are existentially quantified.

A conditional that selects the program (or predicate) p if a condition c holds, and selects q otherwise, is written $p \lhd c \rhd q$, and defined in terms of disjunction and conjunction as follows.

$$(p \lhd c \rhd q) \mathrel{\widehat{=}} c \wedge p \vee \neg\, c \wedge q$$

An (internal) nondeterministic choice $p \sqcap q$ is defined by disjunction.

$$(p \sqcap q) \mathrel{\widehat{=}} p \vee q$$

A variable declaration (**var** x) and a variable undeclaration (**end** x) construct are used to introduce and remove variables from the alphabet. In the case of **var** x, the undecorated variable x is not in its alphabet, but x' is; the alphabet of (**end** x), on other hand, includes x, but not x'. Their definitions are simple.

$$\mathbf{var}\ x \mathrel{\widehat{=}} \exists\, x \bullet \mathbb{I}$$
$$\mathbf{end}\ x \mathrel{\widehat{=}} \exists\, x' \bullet \mathbb{I}$$

The predicate $\mathbb{I}$ is a conjunction of equalities $v' = v$, for all variables v and v' in the alphabet. A standard block (**var** $x \bullet p$) that declares a variable x whose scope is p is defined as **var** $x;\ p;$ **end** x. The definition of independent constructs for declaration and undeclaration of variables is perhaps unusual, but very useful for reasoning. As explained below, they are also central to the definition of parallelism.

Another operator concerned chiefly with the alphabet of predicates is alphabet extension. For a predicate p that does not include x or x' in its alphabet, $p_{+\{x\}}$ is a predicate whose alphabet is that of p with both x and x' added; it does not change the value of the extra observational variable x.

$$p_{+\{x\}} \mathrel{\widehat{=}} p \wedge x' = x$$

In general, this operator can be applied to an arbitrary set of undecorated names; all the variables in the set and their dashed counterparts are added to the alphabet, and all their values are preserved.

Refinement $p \sqsubseteq q$ is characterised by universal reverse implication. Since specifications and programs are all predicates in a UTP theory, they can all be compared directly. In the definition below, square brackets are used as an abbreviated notation for a universal quantification over all variables of the alphabet.

$$p \sqsubseteq q \mathrel{\widehat{=}} [p \Leftarrow q]$$

In words, q is a refinement of p if its behaviour is permitted by p.

More specific theories define operators of more particular interest. For example, the CSP theory provides a definition for parallelism. The main general parallel composition operator is defined with the aid of a restricted parallel operator that combines predicates with disjoint alphabets: it is just conjunction.

$$p \| q \mathrel{\widehat{=}} p \wedge q$$

A more general parallel by merge operator is defined using a merge predicate M that establishes how common variables of the parallel programs should be combined.

$$p \|_M q \mathrel{\widehat{=}} ((p;\ U0(m)_{+A}) \| (q;\ U1(m)_{+B}))_{+m};\ M$$

The set m contains the undashed variables that correspond to the output variables shared by the parallel programs: $m' \mathrel{\widehat{=}} out\alpha P \cap out\alpha Q$. The predicates $U0(m)$ and $U1(m)$ introduce new prefixed variables.

$$U0(m) \mathrel{\widehat{=}} \textbf{var}\ 0.m;\ (0.m = m);\ \textbf{end}\ m$$
$$U1(m) \mathrel{\widehat{=}} \textbf{var}\ 1.m;\ (1.m = m);\ \textbf{end}\ m$$

In $U0(m)$ and $U1(m)$, the new prefixed variables are declared and assigned the value of the corresponding undashed variables. The new variables are declared using the **var** operator and the original variables m are undeclared using the **end** operator. The sets of variables A' and B' are the output alphabets of p and q, but without m'. The predicates $U0(m)_{+A}$ and $U1(m)_{+B}$ establish that the variables not in m are not changed. In $p;\ U0(m)_{+A}$ and $q;\ U1(m)_{+B}$, they retain the output values defined by p and q.

The merge predicate M defines the values of the variables m, in terms of the outputs of the parallel programs in $0.m$ and $1.m$, to reflect the result of their merge. A valid merge predicate should be symmetric on $0.m$ and $1.m$ $((0.m, 1.m := 1.m, 0.m);\ M = M)$; associative $((0.m, 1.m, 2.m := 1.m, 2.m, 0.m);\ M3 = M3$, where $M3$ is a three-way merge relation generated by M); and have no effect when the parallel programs produce the same values $((0.m, 1.m := m, m);\ M = \mathbb{I})$. The CSP parallel composition uses a valid merge function N, that is, $p \|_{CSP} q \mathrel{\widehat{=}} p \|_N q$, where N is defined as follows.

$$N \mathrel{\widehat{=}} \begin{pmatrix} ok' = (0.ok \wedge 1.ok) \wedge wait' = (0.wait \vee 1.wait) \wedge ref' = (0.ref \cup 1.ref) \wedge \\ \exists u \bullet (u \downarrow Ap = 0.tr - tr) \wedge (u \downarrow Aq = 1.tr - tr) \wedge (u \downarrow A(p \| q) = u) \wedge (tr' = tr \frown u) \end{pmatrix};\ Skip$$

The parallel composition diverges if either of the processes diverge $(ok' = (0.ok \wedge 1.ok))$; it terminates if both processes terminate $(wait' = (0.wait \vee 1.wait))$; and if an event is refused by one process, then the parallel composition refuses to engage in the same event $(ref' = (0.ref \cup 1.ref))$. Finally, the parallel processes synchronise on the common events in their alphabet; the sets Ap, Aq and $A(p \| q)$ are the alphabets of the processes p, q and $(p \| q)$, that is, $A(p \| q) = Ap \cup Aq$. It is important to distinguish between, for example, Ap and the alphabet of the predicate p; the set Ap contains the events in which the process p can engage, and the alphabet of the UTP model of p, as already said, includes, ok, $wait$ and others. Traces u produced by the parallel composition are such that $tr' = tr \frown u$. Moreover, when u is restricted to events in $Ap(Aq)$, it results in a valid trace of p (q). Finally, u should be a valid trace of the parallel composition $(u \downarrow A(p \| q) = u)$. The sequence $s \downarrow E$ is that derived from s by restricting it to elements of E only.

In the definition of N, the composition with $Skip$ enforces the arbitrariness of ref'. The definition of $Skip$ uses a healthiness condition R of the theory of reactive systems; it is presented in the next section where we also introduce the corresponding healthiness condition of our timed theory.

$$Skip \mathrel{\widehat{=}} R(\exists ref \bullet \mathbb{I})$$

This process terminates without changing any variable; the quantification over ref makes its value irrelevant, and that of ref' arbitrary. CSP processes do not depend on ref, and, on termination, do not restrict ref'.

$Stop$ is a CSP process that waits forever and does not interact with the environment. Its definition uses two healthiness conditions of the CSP theory: $CSP1$ and $R3$.

$$Stop \mathrel{\widehat{=}} CSP1(ok' \wedge R3(tr' = tr \wedge wait')) \tag{2.1}$$

In the next section, we present our model for *Circus Time*; there we provide additional examples of healthiness conditions, including the definitions of $CSP1$ and $R3$ used above.

2.3. *Circus Time*: semantic model

The alphabet and the healthiness conditions of our theory are similar to those of the *Circus* theory. There are only two differences, which are reflected in some healthiness conditions. The first difference is related to the variables tr and ref. The second difference is related to the program variables as already discussed.

In our theory, interaction with the environment is recorded as a sequence of pairs tr_t (and correspondingly tr'_t), instead of as a sequence of events tr (and correspondingly tr'). Each pair of tr_t and tr'_t records the interaction over a single time unit represented by the sequence index. The first component of the pair is the sequence of events which occurred during the time unit. The second component is the set of events that can be refused at the end of the time unit. There is, therefore, no need for a separate record of the sets of refusals in ref and ref'. Regarding the program variables, to simplify their treatment, they are collected together in a single observation variable $state$ (and the corresponding variable $state'$).

Alphabet The following is a formal description of the observation variables in the alphabet of our theory. As in the UTP theory for CSP, we include boolean variables ok and ok', and $wait$ and $wait'$.

$$ok, ok', wait, wait' : Boolean$$

The variables $state$ and $state'$ are mappings from variable names (in the set N) to values.

$$state, state' : N \rightarrow Value$$

Finally, interaction is captured in the timed traces tr_t and tr'_t

$$tr_t, tr'_t : \mathrm{seq}^{+}(\mathrm{seq}\, Event \times \mathbb{P}\, Event)$$

The set $Event$ contains all possible events of a program. The sequences tr_t and tr'_t are defined to be nonempty using the constructor seq^{+}. We use sequences indexed from 0; the first element represents the initial observations of the program in the state in which it started, and so tr_t and tr'_t include at least that element.

In order to simplify definitions, we introduce the name $trace'$ as an abbreviation for the sequence of events that occurred since the last observation; its value is determined by those of tr'_t and tr_t. In this observation variable we are interested in recording only the events without time.

$$trace' : \mathrm{seq}\, Event$$
$$trace' = Flat(tr'_t) - Flat(tr_t)$$

$Flat$ maps timed traces to untimed traces, as defined below.

$$Flat : \mathrm{seq}^{+}(\mathrm{seq}\, Event \times \mathbb{P}\, Event) \rightarrow \mathrm{seq}\, Event$$
$$Flat(\langle(el, ref)\rangle) = el$$
$$Flat(S \frown \langle(el, ref)\rangle) = Flat(S) \frown el$$

The sequence $s_1 - s_2$ is that obtained by removing from s_1 its prefix s_2 : for instance, $\langle a, b, c\rangle - \langle a, b\rangle = \langle c\rangle$; the subtraction $s_1 - s_2$ is well defined only if s_2 is a prefix of s_1. Our specification of $trace'$ is well defined because the healthiness condition $\mathbf{R1}_t$ of our theory enforces that $Flat(tr_t)$ is a prefix of $Flat(tr'_t)$.

Healthiness conditions There are often intuitive reasons why programs satisfy a given healthiness condition. For example, it would be unusual for a program to make time go backwards or change the history of what happened before it started. *Circus Time* satisfies the healthiness conditions similar to those defined in [HH98] for CSP processes and in [OCW07a] for *Circus*, with some additional considerations regarding time.

The first healthiness condition for *Circus* (and CSP) is $R1$; it requires that a program cannot change the trace tr that occurred before it started. It is defined as $R1(A) \mathrel{\widehat{=}} A \wedge tr \leq tr'$, where $tr \leq tr'$ states that tr is a prefix of tr'. Our corresponding healthiness condition $\mathbf{R1}_t$ states that the timed traces can only be expanded. This means that programs cannot make time go backwards: new traces cannot be shorter.

$$\mathbf{R1}_t(A) \mathrel{\widehat{=}} A \wedge Expands(tr_t, tr'_t)$$

We define a relation $Expands$ between two timed traces as follows.

$$Expands(tr_t, tr'_t) \mathrel{\widehat{=}} (front(tr_t) \leq tr'_t) \wedge (fst(tr_t(\#tr_t))) \leq fst(tr'_t(\#\, tr_t)))$$

We define that tr'_t expands tr_t if, and only if, the front of tr_t is a prefix of tr'_t, and the untimed trace registered at the last time unit of tr_t is a prefix of the trace registered at the same time in tr'_t. The standard operations on sequences

head, *last*, *front*, *tail*, *fst* and *snd* are defined in [She06]. The healthiness condition $\mathbf{R1}_t$ is idempotent and closed over conjunction, disjunction, sequence, and conditional. Detailed proofs of these and other properties of $\mathbf{R1}_t$ can be found in [She06] as well.

The condition $R2$ states that the program execution is independent of the previous interactions with the environment. In the UTP, two definitions for $R2$ are given. The first definition presented below states that starting A after a particular trace tr, and finishing with tr' is the same as starting it after an arbitrary trace s and finishing with this trace as a prefix of the trace $tr' - tr$.

$$R2(A) \mathrel{\widehat{=}} \sqcap_s A[s, s \frown (tr' - tr)/tr, tr']$$

The indexed nondeterministic choice ($\sqcap_s$) captures the fact that the value of s is irrelevant. The second characterisation of $R2$ says that A can alternatively start after an empty trace.

$$R2(A) \mathrel{\widehat{=}} A[\langle\rangle, (tr' - tr)/tr, tr']$$

The equivalence of the two definitions in terms of their characterization of healthy predicates is explored in [CW06]. We define $\mathbf{R2}_t$ in a form similar to that of the second definition, which we believe to be clearer and simpler for reasoning. The initial trace in our model, however, is not $\langle\rangle$, but a trace that contains a single pair composed of an empty sequence of events and an arbitrary refusal set. Consequently, our healthiness condition also imposes that the initial refusal set of a program is also irrelevant for its behaviour. This is an issue that is considered in a separate healthiness condition in the original CSP theory of the UTP.

$$\mathbf{R2}_t(A) \mathrel{\widehat{=}} \exists\, ref \bullet A[\langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t)/tr_t, tr'_t]$$

where *dif* is used to obtain the difference between two timed traces and is defined as follows.

$$dif(tr'_t, tr_t) \mathrel{\widehat{=}} \langle(fst(tr'_t(\#tr_t)) - fst(tr_t(\#tr_t)), snd(tr'_t(\#tr_t)))\rangle \frown tail(tr'_t - front(tr_t))$$

The reason why we use the *dif* function, instead of simply using trace subtraction, is because the initial timed trace tr_t is not necessarily a prefix of the final trace tr'_t. Instead, $\mathbf{R1}_t$ established that they are related by *Expands*. In [She06] we show that $\mathbf{R2}_t$ is idempotent, and closed over conjunction, disjunction, sequence, and conditional. We also show that it is commutative with respect to $\mathbf{R1}_t$.

The healthiness condition $R3$, as defined for *Circus*, assures that a program cannot start in a waiting state: $R3(A) \mathrel{\widehat{=}} \mathbb{I} \lhd wait \rhd A$. If *wait* is *true* (the previous program did not terminate), then the program behaves as $\mathbb{I}$; otherwise, its effect takes place. The program $\mathbb{I}$ preserves the traces if the previous program diverges, otherwise it simply leaves the program variables unchanged. We give a definition for $\mathbb{I}$ below.

$$\mathbb{I} \mathrel{\widehat{=}} (\neg ok \wedge tr \leq tr') \vee (ok' \wedge (tr' = tr) \wedge (wait' = wait) \wedge (state' = state) \wedge (ref = ref'))$$

We define the healthiness condition $\mathbf{R3}_t$ following the same principle; the definition is identical to that of $R3$ except for the use of $\mathbb{I}_t$, which acts on timed traces and uses *Expands*.

$$\mathbf{R3}_t(A) \mathrel{\widehat{=}} \mathbb{I}_t \lhd wait \rhd A$$

The definition of $\mathbb{I}_t$ does not mention explicitly $ref = ref'$ because, in the time model, the refusal information is encoded in the timed traces, so $(tr_t = tr'_t)$ covers the condition that the refusal sets need to be maintained.

$$\mathbb{I}_t = (\neg ok \wedge Expands(tr_t, tr'_t)) \vee (ok' \wedge (tr'_t = tr_t) \wedge (wait' = wait) \wedge (state' = state))$$

The condition $\mathbf{R3}_t$ is idempotent, and closed over conjunction, disjunction, sequence, and conditional [She06].

A reactive program satisfies the condition $\mathbf{R}_t$ defined by the functional composition of $\mathbf{R1}_t$, $\mathbf{R2}_t$ and $\mathbf{R3}_t$.

$$\mathbf{R}_t \mathrel{\widehat{=}} \mathbf{R3}_t \circ \mathbf{R2}_t \circ \mathbf{R1}_t$$

A similar condition R for reactive programs is given in the UTP. The order in which the healthiness conditions are applied in $\mathbf{R}_t$ is irrelevant as they are commutative [She06].

As described in the UTP, a reactive program is also a CSP process if it satisfies five more healthiness conditions. The condition $CSP1$ enforces that, if a program starts in an unstable state, then we only have the guarantee that traces are extended $CSP1(A) \mathrel{\widehat{=}} (\neg ok \wedge tr \leq tr') \vee A$. Again a similar condition is defined for the time model, except that we use the *Expands* relation instead of the trace prefix relation.

$$\mathbf{CSP1}_t(A) \mathrel{\widehat{=}} (\neg ok \wedge Expands(tr_t, tr'_t)) \vee A$$

The healthiness condition $\mathbf{CSP1}_t$ has been shown to be idempotent, closed over conjunction, disjunction, sequence, and conditional, and commutative with $\mathbf{R1}_t$, $\mathbf{R2}_t$ and $\mathbf{R3}_t$ [She06].

The healthiness condition $\mathbf{CSP2}_t$ imposes that a program cannot require nontermination. Therefore, if an observation is valid when ok' is *false*, then it should also be valid if ok' is *true*.

$$\mathbf{CSP2}_t(A) \mathrel{\widehat{=}} A;\ ((ok \Rightarrow ok') \wedge (tr'_t = tr_t) \wedge (wait' = wait) \wedge (state' = state))$$

$\mathbf{CSP2}_t$ is idempotent and closed over disjunction, sequence, and conditional, but not conjunction [She06].

In the *Circus* theory, a condition $CSP3$ requires *Skip* to be the left unit of sequential composition. This imposes that the value of *ref* is irrelevant; as mentioned above, this is already guaranteed by $\mathbf{R2}_t$.

The condition $\mathbf{CSP4}_t$ enforces that *Skip* is the right unit of sequential composition. It is similar to $CSP4$: intuitively, $\mathbf{CSP4}_t$ requires that, on termination or divergence, the value of the refusal set is irrelevant.

$$\mathbf{CSP4}_t(A) \mathrel{\widehat{=}} A;\ Skip$$

The action *Skip* terminates without consuming time; its definition is similar to that in the CSP theory.

$$Skip \mathrel{\widehat{=}} \mathbf{R}_t(\exists\, ref \bullet ref = snd(last(tr_t)) \wedge \mathbb{I}_t)$$

The action *Skip* is $\mathbf{R}_t$ healthy by definition; it is $\mathbf{CSP1}_t$ and $\mathbf{CSP2}_t$ healthy because $\mathbb{I}_t$ is. The condition $\mathbf{CSP4}_t$ is idempotent and closed over disjunction, conditional and sequence, but not conjunction [She06].

Finally $\mathbf{CSP5}_t$ requires that *Skip* is the unit of interleaving. Because *Circus Time* has no interleaving operator, we use the parallel composition operator with an empty synchronisation set.

$$\mathbf{CSP5}_t(A) \mathrel{\widehat{=}} A \llbracket \Delta A \mid \{\!|\!\} \mid \{\} \rrbracket\ Skip$$

The set ΔA contains all the programming variables that the action A can change. This condition enforces that the refusal sets are subset closed: if a particular set of events is a valid refusal set, so are all its subsets.

The UTP parallel operator is based on the alphabet of events of the processes, but in *Circus*, and in CSP, it is based on a synchronisation set: it can only refuse events that are refused by both parallel actions and that are in the synchronisation set, since an event that is refused by one action can be accepted by the other, if it is not in the synchronisation set. In addition, in *Circus*, the parallel operator enforces a partition on the state variables to avoid interference. We call TM the merge predicate that defines parallelism in *Circus*.

$$A \llbracket s_A \mid cs \mid s_B \rrbracket B \mathrel{\widehat{=}} A \parallel_{TM(cs, s_A, s_B)} B$$

We define the merge predicate $TM(cs, s_A, s_B)$ as shown below. It takes as parameters the synchronisation set cs and the sets of variables s_A and s_B that each action can change.

$$TM(cs, s_A, s_B) \mathrel{\widehat{=}} \left(\begin{array}{l} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ dif(tr'_t, tr_t) \in TSync(dif(0.tr_t, tr_t), dif(1.tr_t, tr_t), cs) \wedge \\ state' = (s_B \lhd 0.state) \cup (s_A \lhd 1.state) \end{array} \right)$$

As for the UTP operator, the parallelism diverges if one of the engaged actions diverges, and terminates when both actions terminate. The resulting trace is a member of the set of traces produced by the synchronisation function $TSync$. It takes two timed traces and a set of events on which the actions should synchronise, and yields the set containing all possible traces resulting from the synchronisation. Its definition is equational.

$$\begin{array}{l} TSync(S_1, S_2, cs) = TSync(S_2, S_1, cs) \\ TSync(\langle\rangle, \langle\rangle, cs) = \{\} \\ TSync(\langle(t, r)\rangle, \langle\rangle, cs) = \{\langle(t', r)\rangle \mid t' \in Sync(t, \langle\rangle, cs)\} \\ TSync(\langle(t_1, r_1)\rangle \frown S_1, \langle(t_2, r_2)\rangle \frown S_2, cs) = \\ \quad \{\, \langle(t', r')\rangle \mid t' \in Sync(t_1, t_2, cs) \wedge r' = (((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \backslash cs))\,\} \frown TSync(S_1, S_2, \mathsf{cs}) \end{array}$$

The first equation states that $TSync$ is symmetric. Next, we consider the case in which the two input traces are empty: the result is the empty set. If one of the traces contains a single element, and the other is empty, then the set of synchronisation traces is determined by the results of the synchronisation of the trace element from the non-empty trace with the empty trace; the refusal set is the same in all cases. The synchronisation is defined by the standard function $Sync$ defined for CSP in [Ros98]; it gives the set of all possible combinations of untimed traces, given two untimed traces and a synchronisation set. If both traces are non-empty, then the first elements of the sequences are synchronised. The refusal set at each time instant is determined by the refusals of both actions.

The refusal set is $((r_1 \cup r_2) \cap \mathtt{cs}) \cup ((r_1 \cap r_2) \setminus \mathtt{cs})$, which includes only events that appear in the synchronisation set and are refused by either action, and events that are refused by both actions and do not appear in the synchronisation set. Finally, *state* is defined by combining $0.state$ and $1.state$, after removing from them the variables over which the other action has control. The domain subtraction operator $\lhd$ is used to remove the names in s_B from the domain of $0.state$, and those in s_A from the domain of $1.state$. The sets s_A and s_B partition the state, so that the combination can be achieved by union. In examples, we usually relax this restriction, and use disjoint sets, which can be made into partitions in the obvious way. Some laws of parallelism are presented in Appendix B.

A *Circus Time* action needs to satisfy the condition $\mathbf{CSP}_t$.

$$\mathbf{CSP}_t \mathrel{\widehat{=}} \mathbf{CSP5}_t \circ \mathbf{CSP4}_t \circ \mathbf{CSP2}_t \circ \mathbf{CSP1}_t \circ \mathbf{R}_t$$

The condition $\mathbf{CSP}_t$ includes the condition $\mathbf{R}_t$ for timed reactive action and the four CSP-like conditions.

2.4. *Circus Time*: semantics

In this section we give the semantics of *Circus Time* using the UTP theory presented in the previous section. We use a denotational style, where each construct of *Circus Time* is mapped to a predicate of the UTP theory. Here, we define only the time operators, prefixing, and external choice and hiding, which are used in the definition of timeout. The definitions of the other operators are either similar to those in the UTP theories for CSP and for *Circus* [She06], and can be found in Appendix A, or were introduced in previous sections. The definitions of internal choice, sequence, and conditional are those of the general theory of relations, and were discussed in Sect. 2.2. Recursion is just the least fixed-point operator, which is defined algebraically like in [HH98, p.54]. Our model is a complete lattice with respect to the refinement relation, with *Chaos* at the bottom, and internal choice as the greatest lower bound operator. This follows directly from properties of the healthiness conditions, which are monotonic idempotent functions. *Skip* and parallelism are used as part of the description of the healthiness conditions, and were also discussed previously in Sect. 2.3.

2.4.1. *Wait*

The only possible behaviour for *Wait t* is to wait for the specified number of time units before terminating.

$$Wait\ t \mathrel{\widehat{=}} \mathbf{CSP1}_t(\mathbf{R}_t(ok' \wedge delay(t) \wedge trace' = \langle\rangle))$$

In this definition, we use the predicate $delay(t)$, which is defined as follows.

$$delay(t) \mathrel{\widehat{=}} (wait' \wedge (\#tr'_t - \#tr_t) < t) \vee (\neg wait' \wedge (\#tr'_t - \#tr_t) = t \wedge state = state')$$

The $delay(t)$ predicate requires the program to wait for t time units, and terminate at time t. The delay is directly reflected in the timed trace. The passage of each time unit is captured by the inclusion of a new element in the sequence, but no state change ($state = state'$) nor new communication ($trace' = \langle\rangle$) occurs. Therefore, before t time units have passed, the sequence will have increased no more than t elements; precisely when the process terminates ($\neg wait'$), the sequence will have t new observations. In the next section, we study the relation that exists between *Wait t* and the UTP *Skip*.

The functions $\mathbf{R}_t$ and $\mathbf{CSP1}_t$ are used in the definition of *Wait t* to ensure its healthiness. The other healthiness conditions are already satisfied. $\mathbf{CSP2}_t$ is satisfied because *Wait t* does not require divergence. It also does not restrict the refusals, since its only requirement on tr'_t is based on $trace'$, which ignores the refusal sets, or on the sizes of tr'_t and tr_t, and so *Wait t* is $\mathbf{CSP4}_t$ and $\mathbf{CSP5}_t$ as well.

The action *Wait t* satisfies expected properties already discussed, for example, in [Sch00]. The proofs are presented in [She06]. Some properties are listed in Appendix B.

2.4.2. *Prefixing*

While the program is waiting to communicate on a channel, time can pass and this is registered in the trace as entries $(\langle\rangle, \{x : Event \bullet x \neq c\})$, where c is the channel through which communication is expected. We divide the definition of communication in two parts. The first part is the predicate $wait_com(c)$, which models the state of an action waiting to communicate on channel c. The only possible observation is that the communication channel

cannot appear in the refusal set during the observation period.

$$wait_com(c) \mathrel{\widehat{=}} wait' \wedge possible(tr_t, tr'_t, c) \wedge trace' = \langle\rangle$$

During the waiting period, we may have more than one refusal set; the predicate $possible(tr, tr', c)$ is used to assure that the channel c is not refused in any of the refusal sets.

$$possible(tr_t, tr'_t, c) \mathrel{\widehat{=}} \forall\, i : \#tr_t..\#tr'_t \bullet c \notin snd(tr'_t(i))$$

The final part of the predicate $wait_com(c)$ indicates that, during the waiting period the program will not communicate on other channels, but would allow time to pass ($trace' = \langle\rangle$).

The second part of the definition of communication models the terminating state, and is partially represented by the predicate $term_com(c)$. It reflects the communication of a value e over a channel c. The communication does not take any time ($\#tr' = \#tr$), but the event appears in the traces of the observation.

$$term_com(c) \mathrel{\widehat{=}} \neg wait' \wedge trace' = \langle c\rangle \wedge \#tr'_t = \#tr_t$$

It is important to observe that the above predicate *term_com* only represents the exact moment in which the communication takes place. Terminating observations in the time model are either an initial waiting period followed by termination or an immediate termination. This is expressed as follows.

$$terminating_com(c) \mathrel{\widehat{=}} (wait_com(c);\ term_com(c)) \vee (term_com(c))$$

Finally, we define the communication $c.e \rightarrow Skip$ as shown below.

$$c \rightarrow Skip \mathrel{\widehat{=}} CSP1_t(ok' \wedge R_t(wait_com(c) \vee terminating_com(c)))$$

The semantics of prefixing can be given in terms of communication and sequential composition.

$$com \rightarrow A \mathrel{\widehat{=}} (com \rightarrow Skip);\ A$$

Here *com* stands for a synchronisation c as above, or an input or output, which are defined in Appendix A.

2.4.3. External choice

An external choice $A \Box B$ is determined by the environment; it behaves as either A or B, whichever reacts first to the environment. This is characterised as two possible behaviours: either the choice is in a waiting state, and only internal behaviour acceptable by A and B can take place, or the choice reacts to its environment after waiting for an external event that satisfies either A or B, or both and, in this case, the choice is non-deterministic. The definition in the CSP theory is as follows.

$$A \Box B \mathrel{\widehat{=}} CSP2((A \wedge B) \lhd Stop \rhd (A \vee B))$$

The use of *Stop*, which is a process, as a condition is perhaps surprising, but we observe that CSP processes in the UTP theory for CSP are just predicates, and can be used interchangeably.

Using the definition of conditional, we can rewrite the above definition of external choice as shown below.

$$A \Box B = CSP2(((A \wedge B) \wedge Stop) \vee ((A \vee B) \wedge \neg Stop))$$

The definition of external choice is a disjunction. The first disjunct $((A \wedge B) \wedge Stop)$ defines the behaviour when A and B agree on waiting. The second disjunct $(A \vee B) \wedge \neg Stop)$ defines the state in which the choice is made; it is conditioned by $\neg Stop$. A choice is taken when either A or B diverges or terminates, or when there is a change to the trace recording a reaction of A or B. The healthiness condition $CSP2$ is applied because it is not preserved by conjunction; it ensures that an external choice of healthy actions is healthy.

The resolution of the choice based on divergence or termination is perhaps surprising. In the case of divergence, we observe that external choice is a strict operator: if one of the actions diverges, then the whole choice diverges. Regarding termination, we follow the CSP semantics, since we want *Circus* to be compatible with that notation. In CSP, it is necessary for termination to define an external choice to make sure, for example, that sequence is associative and that *Skip* is the unit of sequence [Ros98].

In *Circus Time*, we take a similar approach, but we need to consider time. Time considerations for an external choice arise in two forms: first, while both actions agree on waiting, we need to track the passage of time. Secondly, we need to eliminate the observations of A and B that do not satisfy the initial waiting conditions. To illustrate this consider the following action $A = Wait\ 2 \Box Wait\ 3$. It makes a choice of either terminating after 2 time

units ($Wait\ 2$) or after three time units ($Wait\ 3$). If we used the CSP definition of external choice above, we could conclude the following.

$$A = CSP2((Wait\ 2 \wedge Wait\ 3) \lhd Stop \rhd (Wait\ 2 \vee Wait\ 3))$$

The predicate $Wait\ 2 \wedge Wait\ 3$ describes the behaviour when both actions agree on waiting. This, however, only occurs in the first 2 time slots; after that, $Wait\ 2$ does not agree on waiting further and terminates. Therefore, the behaviour of A is that of $Wait\ 2$. In the above description, however, the terminating behaviour is defined by $((Wait\ 2 \vee Wait\ 3) \wedge \neg Stop)$. This allows the terminating observation of $Wait\ 2$ after 2 time units, but also the termination of $Wait\ 3$ after 3 time units; this is not admissible in our time theory.

A similar consideration needs to be made regarding external choices involving $Wait$ and communication. To illustrate the problem, we study the behaviour of the action $B = Wait\ 2 \Box (a \rightarrow Skip)$. Both actions in the choice agree on waiting for the first 2 time units, and so, the communication on a can only occur during this period. This external choice acts as a timeout, offering to communicate on a for the first 2 time units, and then forcing a termination. It would not be appropriate to describe the choice as $((Wait\ 2 \vee (a \rightarrow Skip)) \wedge \neg Stop)$, because the behaviour of $(a \rightarrow Skip)$ is not possible after 2 time units.

In view of these considerations, our definition of external choice is as follows.

$$A \Box B \mathrel{\widehat{=}} CSP2_t(ExtChoice1(A, B) \vee ExtChoice2(A, B))$$

As with the CSP definition, we have a disjunction: the disjunct $ExtChoice1(A, B)$ describes the behaviour when both actions are waiting, agree on internal behaviour, and do not interact with the environment.

$$ExtChoice1(A, B) \mathrel{\widehat{=}} (A \wedge B \wedge Stop)$$

The disjunct $ExtChoice2(A, B)$ captures the behaviour in the case A and B do not agree on internal behaviour or on waiting for the environment, by either reacting to an external event or terminating.

$$ExtChoice2(A, B) \mathrel{\widehat{=}} Dif\,Detected(A, B) \wedge (A \vee B)$$

The predicate $DifDetected(A, B)$, rather than $\neg\ Stop$, is used to determine when a choice is made.

$$Dif\,Detected(A, B) \mathrel{\widehat{=}} \neg ok' \vee \left(\begin{array}{l} (ok \wedge \neg wait \wedge ((A \wedge B \wedge ok' \wedge wait' \wedge trace' = \langle\rangle) \vee Skip)); \\ (ok' \wedge ((\neg wait' \wedge tr'_t = tr_t) \vee fst(head(dif(tr_t, tr'_t))) \neq \langle\rangle)) \end{array} \right)$$

The predicate $DifDetected(A, B)$ is a disjunction: the first disjunct captures the behaviour when A or B diverges and, as a consequence, the external choice diverges. The second disjunct defines the situation in which A and B either agree on waiting without interacting with the environment ($ok' \wedge wait' \wedge trace' = \langle\rangle$) or do not agree on waiting at all ($Skip$). In both cases, a predicate in sequence defines that we should then either have a terminating behaviour, if there is no interaction with the environment, or the interaction should take place immediately after the waiting period. A few properties or external choice are listed in Appendix B, and a comprehensive set, with proofs, can be found in [She06].

2.4.4. Hiding

The CSP hiding operator is used to define that some events occur internally and are not recorded in observations. When events are hidden in this way, they occur automatically and instantaneously, as soon as they can: they become urgent. The UTP definition for this operator is as follows.

$$p(tr', ref') \setminus cs \mathrel{\widehat{=}} R(\exists s \bullet p(s, (cs \cup ref')) \wedge L);\ Skip$$

The definition of L is shown below.

$$L \mathrel{\widehat{=}} (tr' - tr) = (s - tr) \downarrow (Ap - cs)$$

The definition of hiding takes an arbitrary valid trace s of the process p, and restricts it to the set $(Ap - cs)$, which contains all the events in the alphabet of p, that is, Ap, except for those that occur in the set of hidden events cs. These are also added to the refusal set $(cs \cup ref')$ of the resulting process. The healthiness condition R and the sequence with $Skip$ guarantee that hiding preserves healthiness.

The definition for the hide operator in the time model is similar.

$$A \setminus cs \mathrel{\widehat{=}} R_t(\exists s_t \bullet A[s_t/tr'_t] \wedge dif(tr'_t, tr_t) = dif(s_t, tr_t) \downarrow_t (Event - cs));\ Skip$$

We use a timed trace restriction operator $\downarrow_t$, which, similarly to the sequence operator, restricts the resulting traces at each time unit to the given set of events cs. Because time traces register the refusals at the end of each time unit, we add the complement of the restricted events ($Events - cs$) to the refusals of the resulting timed trace. The timed trace restriction operator is defined as follows.

$$t_b = t_a \downarrow_t cs \Leftrightarrow \forall i : 1..\#t_a \bullet \left(\begin{array}{l} fst(t_b(i)) = fst(t_a(i)) \downarrow cs \wedge \\ snd(t_a(i)) = (snd(tr_b(i)) \cup (Events - cs)) \wedge \\ \#t_a = \#t_b \end{array} \right)$$

It is important to notice that the timed trace restriction does not change the size of the trace ($\#t_a = \#t_b$). Therefore, the interactions are hidden, but the time that they take is still recorded. Several properties of hiding are listed in Appendix B; many more can be found in [She06].

Like with the definition of hiding in the CSP theory for the UTP, the final *Skip* turns the possibility of divergence into the actual *Chaos* [HH98, p.214]. For example, the semantics of $\mu\ X \bullet a \rightarrow X$ allows ok' to be false, and this lets us conclude that $(\mu\ X \bullet a \rightarrow X) \setminus \{\!|a|\!\} = Chaos$ because of the sequence with *Skip*.

2.4.5. Timeout

In the timeout $A \overset{d}{\rhd} B$, the action A should react within d time units, otherwise B takes over. We define this behaviour using external choice, *Wait*, and hiding, as shown below.

$$A \overset{d}{\rhd} B = (A \Box (Wait\ d;\ int \rightarrow B)) \setminus \{int\}$$

The action $A \overset{d}{\rhd} B$ can only behave as A if it engages on a communication or terminates before the wait period d elapses. The event *int* is fresh: not used by A or B; it is internal (hidden), and so it triggers the external choice and forces it to select the second option once the wait period is finished.

An extensive list of expected properties of timeout have been proved in [She06].

3. Linking models

When studying the features and models of a language, it is important to relate them to other models and languages. A special benefit of using the UTP is that these relationships can be explored in a natural way. Different models are characterised by different alphabets or healthiness conditions. The difference in the expressive power of the models is a motivation to explore the relationships between them: we want to make the most of all models. Studying the relationship that exists between the models makes it possible to create a solid semantic basis for frameworks that use integrated techniques and tools.

In this section, we present a relationship between the *Circus Time* model and the untimed model of *Circus*. An inverse mapping, which links the untimed model to the *Circus Time* model is presented as well. We show that the combination of such links is a *Galois Connection*.

3.1. A conservative mapping L

Our theory was developed to capture time information, but also preserves the untimed semantics of programs. In this section, we explain what we mean by semantic preservation.

To show the relationship between the untimed and the timed theories, we define a function L which, given a predicate of the *Circus Time* theory, defines its behaviour in the original *Circus* theory.

$$L(A) \mathrel{\widehat{=}} \exists tr_t, tr'_t \bullet \left(\begin{array}{l} A \wedge tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{array} \right)$$

The function L maps a predicate A of the timed theory to a predicate in the untimed theory. This is achieved by hiding the timed traces, tr_t and tr'_t, while introducing the untimed observation variables: tr and tr' are obtained

by applying the *Flat* function to the timed traces, and a projection on the second element of their last entries determine the refusal sets ref and ref'. The variables ok, $wait$, and $state$, and their dashed counterparts, are not affected by L. It establishes a very direct correspondence between the predicates.

An important property of L is that it preserves healthiness. For any timed healthiness condition $\mathbf{H}_t$, except only for $\mathbf{R2}_t$, L maps $\mathbf{H}_t$-healthy predicates to H-healthy predicates, where H is a healthiness condition in the untimed theory [She06]. For example, for $\mathbf{R1}_t$, we have the following law.

Law 1 $L(\mathbf{R1}_t(A)) = R1(L(A))$

For $\mathbf{R2}_t$, however, we have the weaker result below.

Law 2 $L(\mathbf{R2}_t(A)) \Rightarrow \exists\, ref \bullet R2(L(A))$

Proof.

$$L(\mathbf{R2}_t(A)) \qquad \text{[definition of } \mathbf{R2}_t\text{]}$$

$$= L(\exists\, ref \bullet A[\langle(\langle\rangle, ref)\rangle, dif(tr'_t, tr_t)/tr_t, tr'_t]) \qquad \text{[predicate calculus]}$$

$$= L(\exists\, ref, ttr, ttr' \bullet A[ttr, ttr'/tr_t, tr'_t] \wedge ttr = \langle(\langle\rangle, ref)\rangle \wedge ttr' = dif(tr'_t, tr_t)) \qquad \text{[definition of } L\text{]}$$

$$= \exists\, tr_t, tr'_t \bullet \left(\begin{array}{l} (\exists\, tref, ttr, ttr' \bullet A[ttr, ttr'/tr_t, tr'_t] \wedge ttr = \langle(\langle\rangle, tref)\rangle \wedge ttr' = dif(tr'_t, tr_t)) \wedge \\ tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{array} \right)$$

[predicate calculus]

$$= \exists\, tref, ttr, ttr' \bullet \left(\begin{array}{l} A[ttr, ttr'/tr_t, tr'_t] \wedge ttr = \langle(\langle\rangle, tref)\rangle \wedge \\ \exists\, tr_t, tr'_t \bullet \left(\begin{array}{l} ttr' = dif(tr'_t, tr_t) \wedge \\ tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right)$$

[definition of *diff* and properties of sequences]

$$= \exists\, tref, ttr, ttr' \bullet \left(\begin{array}{l} A[ttr, ttr'/tr_t, tr'_t] \wedge ttr = \langle(\langle\rangle, tref)\rangle \wedge \\ \exists\, tr_t, tr'_t, p, t, r \bullet \left(\begin{array}{l} tr_t = p \frown \langle(t, r)\rangle \wedge \\ tr'_t = p \frown \langle(t \frown fst(head(ttr')), snd(head(ttr')))\rangle \frown tail(ttr') \wedge \\ tr = Flat(tr_t) \wedge tr' = Flat(tr'_t) \wedge \\ ref = snd(last(tr_t)) \wedge ref' = snd(last(tr'_t)) \end{array} \right) \end{array} \right)$$

[predicate calculus, and properties of *Flat* and sequences]

$$= \exists\, tref, ttr, ttr' \bullet \left(\begin{array}{l} A[ttr, ttr'/tr_t, tr'_t] \wedge ttr = \langle(\langle\rangle, tref)\rangle \wedge \\ \exists\, p, t \bullet tr = Flat(p) \frown t \wedge tr' = Flat(p) \frown t \frown Flat(ttr') \wedge ref' = snd(last(ttr')) \end{array} \right)$$

[predicate calculus and properties of sequences]

$$= \exists\, ref, tr_t, tr'_t \bullet A \wedge tr_t = \langle(\langle\rangle, ref)\rangle \wedge Flat(tr'_t) = tr' - tr \wedge ref' = snd(last(tr'_t))$$

[properties of sequences]

$$\Rightarrow \exists\, ref, tr_t, tr'_t \bullet \left(\begin{array}{l} A \wedge Flat(tr_t) = \langle\rangle \wedge ref = snd(last(tr_t)) \wedge \\ Flat(tr'_t) = tr' - tr \wedge ref' = snd(last(tr'_t)) \end{array} \right) \qquad \text{[property of substitution]}$$

$$= \exists\, ref \bullet \exists\, tr_t, tr'_t \bullet \left(\begin{array}{l} A \wedge Flat(tr_t) = tr \wedge ref = snd(last(tr_t)) \wedge \\ Flat(tr'_t) = tr' \wedge ref' = snd(last(tr'_t)) \end{array} \right) [\langle\rangle, tr' - tr/tr, tr']$$

[definitions of L and $R2$]

$$= \exists\, ref \bullet R2(L(A)) \qquad \square$$

As we mentioned previously, $\mathbf{R2}_t$ enforces independence from both history of interaction and initial refusals. Therefore, in the untimed model, it does not correspond directly to $R2$; it covers $CSP3$ as well.

By applying L to *Circus Time* constructs, we obtain corresponding programs in the untimed theory. An important observation is that $L(A)$ does not change the untimed behaviour of A. In fact, L preserves all the basic actions, except *Wait t*; for example, for an assignment, we have the law below.

Law 3 $L([\![x := e]\!]_{time}) = [\![x := e]\!]$

Because the syntax of *Circus Time* and *Circus* are similar, we use the notation $[\![A]\!]_{time}$ to represent the action A in the time theory, and $[\![A]\!]$ to stand for the same action in the untimed theory. In addition, L distributes over all programming constructs [She06]; for example, for a conditional, we have the law below.

Law 4 $L([\![A \lhd b \rhd B]\!]_{time}) = [\![L(A) \lhd b \rhd L(B)]\!]$

For *Wait t*, by hiding the time information in the program that waits for a determined amount of time, L gives a program that can either wait forever (*Stop*) or terminate immediately (*Skip*).

Law 5 $L([\![Wait\ t]\!]_{time}) = [\![Stop]\!] \sqcap [\![Skip]\!]$

Proof.

$$L([\![Wait\ t]\!]_{time}) \qquad \text{[definition of } Wait\ t\text{]}$$

$$= L(\mathbf{CSP1}_t(\mathbf{R}_t(ok' \wedge delay(t) \wedge trace' = \langle\rangle))) \qquad \text{[properties of } L\text{, definition of } \mathbf{R}_t\text{]}$$

$$= CSP1(R1(R3(L(\mathbf{R2}_t(ok' \wedge delay(t) \wedge trace' = \langle\rangle))))) \qquad \text{[as shown in the proof of Law 2]}$$

$$= CSP1\left(R1\left(R3\left(\exists\, ref, tr_t, tr'_t \bullet \left(\begin{array}{l} ok' \wedge delay(t) \wedge trace' = \langle\rangle \wedge \\ tr_t = \langle(\langle\rangle, ref)\rangle \wedge Flat(tr'_t) = tr' - tr \wedge ref' = snd(last(tr'_t)) \end{array}\right)\right)\right)\right)$$

[definitions of *delay* and *trace*]

$$= CSP1\left(R1\left(R3\left(\exists\, ref, tr_t, tr'_t \bullet \left(\begin{array}{l} ok' \wedge \\ \left(\begin{array}{l} wait' \wedge (\#tr'_t - \#tr_t) < t \vee \\ \neg wait' \wedge (\#tr'_t - \#tr_t) = t \wedge state' = state \end{array}\right) \wedge \\ Flat(tr'_t) - Flat(tr_t) = \langle\rangle \wedge \\ tr_t = \langle(\langle\rangle, tref)\rangle \wedge Flat(tr'_t) = tr' - tr \wedge ref' = snd(last(tr'_t)) \end{array}\right)\right)\right)\right)$$

[predicate calculus and properties of sequences]

$$= CSP1\left(R1\left(R3\left(\exists\, tr'_t \bullet \left(\begin{array}{l} ok' \wedge \left(\begin{array}{l} wait' \wedge (\#tr'_t - 1) < t \vee \\ \neg wait' \wedge (\#tr'_t - 1) = t \wedge state' = state \end{array}\right) \wedge \\ tr' = tr \wedge Flat(tr'_t) = \langle\rangle \wedge ref' = snd(last(tr'_t)) \end{array}\right)\right)\right)\right)$$

[properties of sequences]

$$= CSP1(R1(R3(ok' \wedge tr' = tr \wedge (wait' \vee \neg\, wait' \wedge state = state')))) \qquad \text{[distribution of healthiness conditions over disjunction]}$$

$$= CSP1(R1(R3(ok' \wedge tr' = tr \wedge wait'))) \vee CSP1(R1(R3(\neg\, wait' \wedge state = state'))) \qquad \text{[definition of } R1 \text{ and properties of sequences]}$$

$$= CSP1(R3(ok' \wedge tr' = tr \wedge wait')) \vee CSP1(R1(R3(\neg\, wait' \wedge state = state'))) \qquad [CSP1(R3(ok' \wedge p)) = CSP1(ok' \wedge R3(p))]$$

$$= CSP1(ok' \wedge R3(tr' = tr \wedge wait')) \vee CSP1(R1(R3(\neg\, wait' \wedge state = state'))) \qquad \text{[definition of } Stop\text{]}$$

$$= Stop \vee CSP1(R1(R3(\neg\, wait' \wedge state = state'))) \qquad \text{[definition of } R2 \text{ and commutativity of healthiness conditions]}$$

$$= Stop \vee R(CSP1(\neg\, wait' \wedge state = state')) \qquad \text{[definition of } CSP1\text{]}$$

$$= Stop \vee R(\neg\, ok \wedge tr \leq tr' \vee \neg\, wait' \wedge state = state') \qquad \text{[definition of } R1 \text{ and properties of sequences]}$$

$$= Stop \vee R(\neg\, ok \vee \neg\, wait' \wedge state = state') \qquad \text{[property of } Skip \text{ [CW06] and definition of choice]}$$

$$= [\![Stop]\!] \sqcap [\![Skip]\!] \qquad \square$$

For a timeout action, L is consistent with its characterisation in terms of a wait command.

3.2. The inverse mapping R

The function L is an abstraction; $L(A)$ hides time information and gives a weaker representation of A in the untimed theory. As a result, $L(A)$ can only give a best approximation of the meaning of A, and there might not be an exact inverse of L. It is possible, however, to find a function R which as far as possible undoes the effect of L. Given an untimed predicate B, R gives the weakest timed predicate with the same behaviour.

$$R(B) \mathrel{\widehat{=}} \sqcap \{A \mid L(A) \sqsupseteq B\}$$

Because R is a weak inverse of L, then there is an unavoidable loss of information when applying R to the result of an application of L. The following theorem captures this fact.

Theorem 1 $A \sqsupseteq R(L(A))$

Proof.

$$\begin{aligned}
& R(L(X)) && \text{[definition of } R\text{]} \\
& = \sqcap \{A \mid L(A) \sqsupseteq L(X)\} && [L \text{ is monotonic, and property of greatest lower bound}] \\
& \sqsupseteq \sqcap \{A \mid A \sqsupseteq X\} && \text{[property of greatest lower bound]} \\
& = X && \square
\end{aligned}$$

If we apply the weakening function R to a predicate B and then apply the strengthening function L to the result, this may yield a predicate stronger than B in the untimed theory, as established below.

Theorem 2 $L(R(B)) \sqsupseteq B$

Proof.

$$\begin{aligned}
& L(R(B)) && \text{[definition of } R\text{]} \\
& = L(\sqcap \{A \mid L(A) \sqsupseteq B\} && [L \text{ is monotonic}] \\
& \sqsupseteq \sqcap \{L(A) \mid L(A) \sqsupseteq B\} && \text{[property of greatest lower bound]} \\
& \sqsupseteq B && \square
\end{aligned}$$

These results mean that the functions L and R form a *Galois connection*.

This permits us to explore some properties of the timed language. In particular, we are able to characterise predicates of the timed theory that do not impose any time requirements.

Definition 1 A predicate A is time insensitive if $R(L(A)) = A$.

The above definition states that if, by applying the abstraction function L to a predicate A, and then applying the weak inverse function R to the result, we obtain the same initial predicate A, then the time information in the original predicate A is irrelevant. An example is the action *Stop*. It waits forever and permits time to pass; however, it is time insensitive in the sense that it does not impose any restriction on the time passage. Therefore, removing the time information from *Stop* and then adding arbitrary time records results in the same action *Stop*. On other hand, if we consider *Wait* 3, we observe the following.

$$\begin{aligned}
& R(L(Wait\ 3)) && \text{[definition of } R\text{]} \\
& = \sqcap \{A \mid L(A) \sqsupseteq L(Wait\ 3)\} && \text{[Law 5]} \\
& = \sqcap \{A \mid L(A) \sqsupseteq (Stop \sqcap Skip)\} && \text{[properties of } \sqsubseteq \text{ and definition of } L\text{]} \\
& = \sqcap \{A \mid A = Skip \vee A = Stop \vee \exists n \bullet A = Wait\ n\} \\
& \neq Wait\ 3
\end{aligned}$$

From the above inequality we see clearly, as expected, that the action *Wait* 3 is not time insensitive.

The relationship between the UTP theories permits us to explore properties of timed programs that can be expressed in the untimed theory. The parts of a program that are not time sensitive can be identified and explored, and safety properties of time sensitive programs can still be validated in the untimed theory.

Properties of R with respect to each of the action constructs remain to be formulated. The Galois connection and the laws of L presented in the previous section, however, give a strong indication that the derivation of a rich set of laws is not difficult. An additional property that is central to the argument of soundness of our validation framework is discussed in Sect. 4.5.

3.3. Non-conservative mapping

As discussed above, the abstraction function L, when applied to *Wait d*, gives a nondeterministic choice between *Skip* and *Stop*. This actually introduces a deadlock state into the program, and therefore liveness properties cannot be explored after the application of this abstraction function. The program that results from the application of L may deadlock, even when the original program does not.

A more suitable abstraction for reasoning about liveness would substitute *Wait d* with *Skip*. For that, a new mapping can be defined using L; it ensures that the only possible waiting state for a program is a waiting state that can wait forever, that is, deadlock. The definition of this function $\hat{L}$ is as follows.

$$\hat{L}(A) \mathrel{\widehat{=}} L(A) \wedge (wait' \Rightarrow \forall n \bullet \exists tr_o \bullet tr_o = tr_t \frown \langle(\langle\rangle, ref)\rangle^n \wedge A[tr_o/tr'_t])$$

The notation $\langle e\rangle^n$ stands for a sequence with n occurrences of e. With the above definition, we state that the traces and refusals of $\hat{L}(A)$ are defined from tr_t and tr'_t in the same way as they were for $L(A)$, and also if $wait'$ holds, then it is because there are arbitrarily long traces tr_o of A that record only passage of time: no communications or change in the refusals *ref* (as defined by $L(A)$) take place.

For this new function, we have $\hat{L}([\![Wait\ d]\!]_{time}) = [\![Skip]\!]$. On other hand, the function L is conservative: it preserves the behaviour of the original program as indicated by the Laws 1–5 in the previous section, and other distribution laws in [She06]. The function $\hat{L}$ is not conservative as it does not distribute over parallel composition. To explain what happens, we consider the following example.

$$\begin{aligned} A &\mathrel{\widehat{=}} (a \rightarrow Skip) \overset{2}{\rhd} Skip \\ B &\mathrel{\widehat{=}} Wait\ 3;\ (a \rightarrow Skip) \\ C &\mathrel{\widehat{=}} A \llbracket \{\} \mid \{\!|\ a\ |\!\} \mid \{\} \rrbracket B \end{aligned}$$

We observe that $\hat{L}(C) = \hat{L}(Stop) = Stop$ because the action A times out before the action B is ready to synchronise on the event a. On other hand, $\hat{L}(A) \llbracket \{\} \mid \{\!|a|\!\} \mid \{\} \rrbracket \hat{L}(B)$ is the following action.

$$\begin{aligned} &(((a \rightarrow Skip) \Box (Skip;\ int \rightarrow Skip))\backslash\{int\}) \llbracket \{\} \mid \{\!|\ a\ |\!\} \mid \{\} \rrbracket (Skip;\ (a \rightarrow Skip)) \\ &\quad = (((a \rightarrow Skip) \Box (int \rightarrow Skip))\backslash\{int\}) \llbracket \{\} \mid \{\!|\ a\ |\!\} \mid \{\} \rrbracket (a \rightarrow Skip) \\ &\quad = ((a \rightarrow Skip) \sqcap Skip) \llbracket \{\} \mid \{\!|\ a\ |\!\} \mid \{\} \rrbracket (a \rightarrow Skip) \\ &\quad = (a \rightarrow Skip) \sqcap Stop \end{aligned}$$

In this parallelism, a may occur immediately, and in this case progress is made.

The two mappings L and $\hat{L}$ capture the two extreme observations of the timed model. The first replaces a quantified wait by the option to either wait forever or terminate immediately. The second mapping replaces the quantified wait with an immediate termination; it does not allow the program to wait. Using the first mapping, we obtain corresponding untimed programs that are suitable for analysis of properties of traces, while the second mapping yields results adequate for analysis of liveness properties.

The validation framework that we present in the next section can be used with either L or $\hat{L}$; our discussions, however, are based on L. In addition, we observe that some properties can only be proven in the time model because they are related directly to the timing requirements of the system. Deadlock freedom, for example, cannot be established after the application of L or $\hat{L}$ because synchronisation is time dependent. With $\hat{L}$, at least, if we prove that the untimed program deadlocks, then we know that the timed program does as well. On other hand,

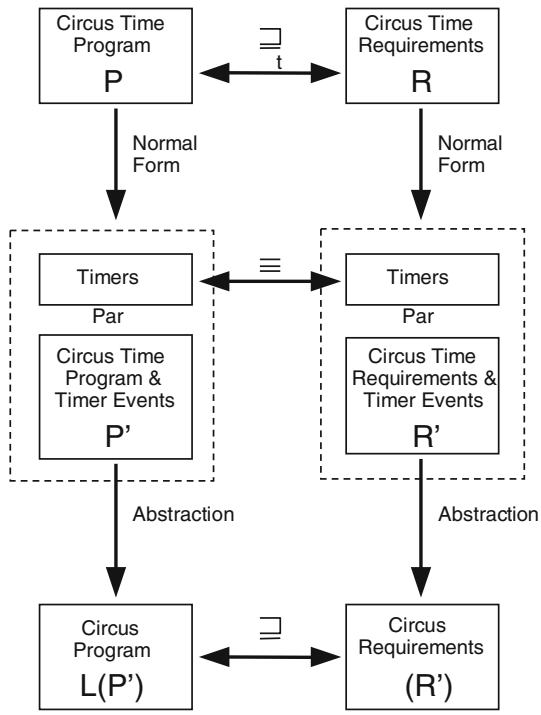

Fig. 2. A heterogeneous framework for analysis of timed programs in *Circus*

if the untimed program does not deadlock, it may still be the case that the timed program deadlocks due to the timing restrictions.

In [Sch00], Schneider presents a notion of timewise refinement to relate Timed CSP programs to untimed CSP specifications. A stepwise development process is proposed, where the early steps are based on untimed programs that describe behaviour without any time constraints. In subsequent steps, time is introduced and the timewise refinement relation is used to ensure that correctness with respect to the untimed properties is preserved. What we propose in the sequel has a different objective: given that we already have a timed program, we want to use techniques of the untimed notation and theory to prove properties.

Mathematically, we define functions like L and $\hat{L}$ that map timed to untimed models, and compare predicates of the different theories only indirectly, using the Galois connection defined by the functions. Schneider, on the hand, introduces new refinement relations that compare timed and untimed programs; as for standard CSP, there are relations based on traces, failures, and failures-divergences. We, of course, work with the UTP theory and its refinement relation, which is closest (though not isomorphic) to the failures-divergences model. In both approaches, it is possible to compare timed and untimed programs; in particular, Schneider's timewise refinement relations give results closer to those obtained with our $\hat{L}$ function. A precise comparison is not trivial, since the models are different. The validation framework that we now describe, in addition, can be used with any Galois connection between the timed and untimed UTP theories.

4. Validation framework

In this section, we present a framework for specification and validation of real-time programs using the timed and untimed theories of *Circus*. It uses the timed theory for specification, and the untimed theory to validate system requirements. A normal form for a timed program is used to obtain a corresponding untimed program that includes special timer events and can be verified to meet time requirements. In this way, we can reason about time properties without making explicit use of time.

Figure 2 illustrates the steps for using the framework, which can be summarised as follows.

1. We start with a program P and a time requirement R, both defined using *Circus Time*. The designer gives a complete description of the system and uses the same notation to describe the desired properties.

2. With the help of the function Φ, defined in the next section, we obtain a normal form program that has the same semantics as the original program (Theorem 3). The normal form is composed of two parts: a set of interleaved timers and a program $P' = \Phi(P)$ with no time operators, but containing internal timer events. The structure of the normal form of the requirement R is the same. Furthermore, the validation requires that the timers of the normal forms of P and R are equivalent. In practice, adjustments may be needed to the normal form of the specification to add extra timers that may be used in the implementation. The presence in the abstract model of timers that are not used does not affect its behaviour. Our normal form, however, is not appropriate for recursive programs that involve the time operators.
3. Next, we apply the function L, or any other abstraction function, to the program P' and to the requirements R'. Although we illustrate our framework with L, since we explore the validation of safety properties, the framework is actually parametrised by the mapping function. For proving liveness properties, the function $\hat{L}$ can be used, instead. Other mapping functions can also be defined.
4. Finally, we show that the untimed program $L(P')$ satisfies the untimed requirements $L(R')$. This guarantees that the timed program P satisfies the timed requirements R (Theorem 5).

In Sect. 4.1, we introduce the normal form, including a definition for timers, and a normalisation strategy. In Sect. 4.2, we introduce variations of the external choice and parallel composition operators to capture the special treatment of the timer events. The function Φ is defined in Sect. 4.3. We prove the correctness of the normalisation strategy in Sect. 4.4. Finally, Sect. 4.5 considers refinement of programs in the normal form, and links it to the refinement relation in the untimed theory.

4.1. The normal form

Usually, timed programs are implemented with timers: the system clock or a dedicated timer. Following this idea, we define a normal form for *Circus Time* actions: a parallel composition of a set of timers and an untimed program that synchronises on the timer events. Semantically, these events have a specialised behaviour when they appear in an external choice or in a parallel composition. Therefore, strictly, the language used to express the normal form is an extension of *Circus Time* with the timer events.

More precisely, a normalised *Circus Time* action takes the following form.

$$\Phi(A)\ \mathbf{par}\ \mathit{Timers}(k, n)$$

The normalisation function Φ maps an arbitrary *Circus Time* action to an action that relies on timer events to synchronise with timers, and does not include any of the time operators. These are confined to the other component of the normal form, $\mathit{Timers}(k, n)$, which is a set of interleaved timer actions $\mathit{Timer}(i)$ with indices i from k to n. The **par** operator is defined in terms of the parallel composition operator, but deals explicitly with termination of the synchronisation between $\Phi(A)$ and $\mathit{Timers}(k, n)$. We use indices k to n for the timers because each normalised action has its own set of timers. To combine them, we need to avoid clashes between their timers, and so we assign arbitrary disjoint intervals to the indices.
The following is the specification of an indexed timer i, which provides a delay of d time units.

$$\mathit{Timer}(i) \mathrel{\widehat{=}} \left(\mu X \bullet \left(\begin{array}{l} \mathit{setup}.i?d \rightarrow \left(\begin{array}{l} (\mathit{halt}.i \rightarrow X) \\ \Box\ \mathit{Wait}\ d;\ ((\mathit{out}.i.d \rightarrow X) \Box (\mathit{terminate}.i \rightarrow \mathit{Skip})) \\ \Box\ (\mathit{terminate}.i \rightarrow \mathit{Skip}) \end{array} \right) \\ \Box\ (\mathit{terminate}.i \rightarrow \mathit{Skip}) \end{array} \right) \right)$$

The timer is initiated with the event $\mathit{setup}.i.d$; it then offers the event $\mathit{halt}.i$ while it waits for d time units; at the end, the event $\mathit{out}.i.d$ is also offered. When either $\mathit{halt}.i$ or $\mathit{out}.i.d$ takes place, the timer is reset. The timer always offers the event $\mathit{terminate}.i$, which terminates its execution. The reason to use the delay d as a parameter to the events *setup* and *out* is that, in an external choice or in a parallelism, these events present a special behaviour that is sensitive to the delay, as explained in detail in the next section. In addition, we observe that the *setup*, *out*, *halt*, and *terminate* events are internal in the normalised system. This means that they are urgent when they become available, and so the time limits defined by the delays are enforced.

$\mathit{Timers}(k, n)$ is the interleaving of the needed timers: one for each time operator.

$$\mathit{Timers}(k, n) \mathrel{\widehat{=}} \|\ i : k \mathinner{..} n \bullet \mathit{Timer}(i)$$

For conciseness, we use the interleaving operator directly, which, as already mentioned, can be expressed in terms

of the parallel operator with an empty synchronisation set. The timers do not change the state, and so they do not need access to any of its variables. In the interleaving above the name set associated with each timer is empty. For simplicity we omit the empty name sets.

The special parallel composition operator **par** applies to an arbitrary action A and to an interleaving of timers $Timers(k, n)$; it is defined as follows.

$$A \ \mathbf{par}\ Timers(k, n) \mathrel{\widehat{=}} (A;\ Terminate(k, n) \llbracket s_A \mid TSet \mid \{\} \rrbracket Timers(k, n)) \backslash TSet$$

The set s_A contains all the variables changed by A. $Terminate(k, n)$ assures that, if A terminates normally, then all the timers are terminated. This is necessary to terminate the parallelism; otherwise the timers do not allow the parallel composition to terminate. $Terminate(k, n)$ is defined as follows.

$$Terminate(k, n) \mathrel{\widehat{=}} \|\!|\ i : k \mathinner{..} n \bullet terminate.i \rightarrow Skip$$

$TSet$ is the set of events that the timer can engage on. Particularly, the events *setup* and *out* have a special behaviour captured by a new external choice and a new parallel composition operator defined in the next section; the set *timerEvents* is used when we need to refer only to these events.

$$timerEvents \mathrel{\widehat{=}} \bigcup \{ i : k \mathinner{..} n;\ d : \mathbb{Z} \bullet \{ setup.i.d, out.i.d \} \}$$

$$TSet = timerEvents \cup \bigcup \{ i : k \mathinner{..} n \bullet \{ halt.i, terminate.i \} \}$$

The operator **par** treats these events as internal. The lemma below establishes that termination of A is handled appropriately: $Terminate(k, n)\ \mathbf{par}\ Timers(k, n)$ terminates immediately.

Lemma 1 $(Terminate(k, n) \llbracket s_A \mid TSet \mid \{\} \rrbracket Timers(k, n)) \backslash TSet = Skip$

Proof.

$$(Terminate(k, n) \llbracket s_A \mid TSet \mid \{\} \rrbracket Timers(k, n)) \backslash TSet \qquad \text{[definition of } Terminate(k, n)\text{]}$$

$$= ((\|\!|\ i : k \mathinner{..} n \bullet terminate.i \rightarrow Skip) \llbracket s_A \mid TSet \mid \{\} \rrbracket Timers(k, n)) \backslash TSet$$

[definition of $Timers(k, n)$]

$$= ((\|\!|\ i : k \mathinner{..} n \bullet terminate.i \rightarrow Skip) \llbracket s_A \mid TSet \mid \{\} \rrbracket (\|\!|\ i : k \mathinner{..} n \bullet Timer(i))) \backslash TSet$$

[definition of $Timer(i)$ and fixed point]

$$= \left(\begin{array}{l} (\|\!|\ i : k \mathinner{..} n \bullet terminate.i \rightarrow Skip) \\ \quad \llbracket s_A \mid TSet \mid \{\} \rrbracket \\ \|\!|\ i : k \mathinner{..} n \bullet \left(\mu X \bullet \left(\left(\begin{array}{l} setup.i?d \rightarrow \\ \left(\begin{array}{l} (halt.i \rightarrow X) \\ \Box \left(\begin{array}{l} Wait\ d; \left(\begin{array}{l} (out.i.d \rightarrow X) \\ \Box \\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \right) \end{array} \right) \backslash TSet$$

[step laws (Law 22 and 21)]

$$= (\|\!|\ i : k \mathinner{..} n \bullet terminate.i \rightarrow Skip) \backslash TSet$$

[$terminate.i \in TSet$, property of hiding (Law 25), and unit of $\|\!|$]

$$= Skip \qquad \Box$$

Before presenting the normal form reduction process in detail, to provide some intuition, we consider the alarm controller that we introduced in Sect. 2.1. The corresponding untimed program is presented below.

$$Disable_{NF} = disable \rightarrow Skip$$
$$Running_{NF} = Disable_{NF} \ \Box\ (disturbed \rightarrow Active_{NF})$$
$$Active_{NF} = setup.2!T_2 \rightarrow ((Disable_{NF};\ halt.2 \rightarrow Skip) \ \Box\ (out.2.T_2 \rightarrow alarm \rightarrow Disable_{NF}))$$
$$Alarm_{NF} = \mu X \bullet enable \rightarrow setup.1!T_1 \rightarrow \left(\begin{array}{l} (Disable_{NF};\ halt.1 \rightarrow Skip) \\ \Box \\ (out.1.T_1 \rightarrow Running_{NF}) \end{array} \right);\ X$$

The generated program contains no time information: just timer events. There is one timer for each timeout

operator used in the original program: the two timers below.

$$\begin{array}{lcl} Timer_1 & = & Timer(1) \\ Timer_2 & = & Timer(2) \end{array}$$

The normal form of the process *Alarm* (in Sect. 2.1) is given by

$$((Alarm_{NF};\ Terminate(1,2)) \llbracket \{\} \mid TSet \mid \{\} \rrbracket (Timer_1 \lll Timer_2)) \setminus TSet$$

In the normalisation, *Disable* and *Running* are not changed, since they do not involve time constructs directly. $Active_{NF}$, on other hand, is obtained from $Active = Disable \overset{T_2}{\rhd} (alarm \rightarrow Disable)$ by replacing the timeout operator with timer events. The event $setup.2.T_2$ sets the second timer to wait for T_2 time units. If $Active_{NF}$ engages on *disable* before T_2 time units, then $Active_{NF}$ and $Timer_2$ synchronise on $halt.2$, and $Active_{NF}$ terminates; $Timer_2$ is then reset. After T_2 time units, $Timer_2$ and, therefore, $Active_{NF}$ synchronise on $out.2.T_2$, and $Active_{NF}$ becomes ready to engage on *alarm*. The *Alarm* action includes another timeout operator that is normalised in an analogous way, but using a separate timer ($Timer_1$).

We observe that our example involves a recursion whose body involves timed operators. Although our technique does not cover recursion in general, in this case the recursion does not cause problems. They arise, for example, when the recursion spawns independent (interleaved) timed actions, each requiring its own timer. In such cases, the normalisation as presented here creates a single timer and is not appropriate. Another source of problems is the fact that each timer is associated with a single operator and recursion can create dynamic dependencies between them. As already said, in our example, the presence of recursion raises no difficulties, but we leave the formal treatment of recursion as future work.

Before presenting the normalisation strategy, which is an entirely algebraic process, we motivate and define two new operators used at intermediate stages of the reduction process.

4.2. Normal form operators

In this section, we present variations of the choice and parallel composition operators. They capture the special behaviour attached to the timer events. Both their UTP definition and algebraic laws are presented.

4.2.1. Normal form choice

To explain why we need a new choice operator and why the timer events need to be treated in a special way, we consider the following example: $(Wait\ 5) \Box (a \rightarrow Skip)$. The normalised action corresponding to *Wait* 5, that is, $\Phi(Wait\ 5)$, is $setup.i.5 \rightarrow out.i.5 \rightarrow Skip$. If we defined Φ to distribute over external choice, the result of applying Φ to our example would be $((setup.i.5 \rightarrow out.i.5 \rightarrow Skip) \Box (a \rightarrow Skip))$. This is not appropriate because the event *setup* should be used only to start the timer and the choice should not be resolved by a communication on *setup*. The desired behaviour is that $setup.i.5$ occurs first and then the choice is made over a or out. In other words, the desired behaviour is $setup.i.5 \rightarrow ((out.i.5 \rightarrow Skip) \Box (a \rightarrow Skip))$. To keep the definition of Φ compositional, and yet get the choice right, we define a new choice operator $\boxtimes$.

We give the semantics of our new choice using the UTP parallel by merge. To justify that, we give below an alternative definition for the conventional choice; it is equivalent to the standard semantics [She06].

$$A \Box B \mathrel{\widehat{=}} CSP2\left(CSP1\left((A \|_{CM} B) \vee \left(\begin{array}{l} ok' \wedge wait \wedge \\ (tr' = tr) \wedge (wait' = wait) \wedge (ref' = ref)(state' = state) \end{array}\right)\right)\right)$$

CM is a choice parallel merge predicate defined as follows.

$$CM \mathrel{\widehat{=}} \left(\begin{array}{l} (0.ok, 0.wait, 0.tr - tr, 0.ref, 0.state), \\ (1.ok, 1.wait, 1.tr - tr, 1.ref, 1.state) \end{array}\right) ChSynch\,(ok', wait', tr' - tr, ref', state')$$

The relation *ChSynch* maps the prefixed variables of the actions A and B to the corresponding dashed observation variables, but in the case of tr' (and $0.tr$ and $1.tr$), it is concerned only with the new interactions that are not

already recorded in tr, that is, $0.tr - tr$, $1.tr - tr$, and $tr' - tr$.

$$\left(\begin{array}{l}(0.ok, 0.wait, 0.tr, 0.ref, 0.state),\\ (1.ok, 1.wait, 1.tr, 1.ref, 1.state)\end{array}\right) ChSynch\,(ok', wait', tr', ref', state') \Leftrightarrow \left(\begin{array}{l}Diverge \vee\\ WaitokNoEvent \vee\\ DoAnyEvent \vee\\ Terminate\end{array}\right)$$

The predicate *Diverge* captures the diverging behaviour: the choice operator can diverge if one of the actions involved in the choice diverges, and that action is selected.

$$Diverge \widehat{=} ok \wedge \neg ok' \wedge \left(\begin{array}{l}(0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr = tr' \wedge 0.ref = ref' \wedge 0.state = state') \vee\\ (1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr = tr' \wedge 1.ref = ref' \wedge 1.state = state')\end{array}\right)$$

Next, we consider the waiting state of the external choice. In an external choice, a waiting state is only observed if both actions agree on waiting and no external events are observed.

$$WaitokNoEvent \widehat{=} \neg wait \wedge \left(\begin{array}{l}ok' \wedge 0.ok = ok' \wedge 1.ok = ok' \wedge\\ wait' \wedge 0.wait = wait' \wedge 1.wait = wait' \wedge\\ tr' = \langle\rangle \wedge 0.tr = tr' \wedge 1.tr = tr' \wedge\\ 0.ref = ref' \wedge 1.ref = ref' \wedge 0.state = state' \wedge 1.state = state'\end{array}\right)$$

This reflects only the non-diverging behaviour, since the diverging behaviour is captured by *Diverge*. Next we consider the situation is which an event is observed on the output trace indicating that a communication took place. In such case, the action that can observe the same event on the head of its traces is chosen.

$$DoAnyEvent \widehat{=} ok \wedge \left(\begin{array}{l}tr' \neq \langle\rangle \wedge\\ \left(\begin{array}{l}(0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr = tr' \wedge 0.ref = ref' \wedge 0.state = state') \vee\\ (1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr = tr' \wedge 1.ref = ref' \wedge 1.state = state')\end{array}\right)\end{array}\right)$$

Finally, if any of the two actions terminates with no communication observed on the outputs, then the terminating action is chosen. This is expressed in the following predicate:

$$Terminate \widehat{=} ok \wedge \left(\begin{array}{l}\neg wait' \wedge\\ \left(\begin{array}{l}(0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr = tr' \wedge 0.ref = ref' \wedge 0.state = state') \vee\\ (1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr = tr' \wedge 1.ref = ref' \wedge 1.state = state')\end{array}\right)\end{array}\right)$$

In a similar way, we give the definition of the semantics of the special choice operator $\boxtimes$.

$$A \boxtimes B \widehat{=} \mathbf{CSP2}_t \left(\mathbf{CSP1}_t \left(A \|_{TCM} B \vee \left(\begin{array}{l}ok' \wedge wait \wedge\\ (tr'_t = tr_t) \wedge (wait' = wait) \wedge (state' = state)\end{array}\right) \right) \right)$$

TCM is a new merge predicate that embeds the semantics of the new choice operator. The relation *TChSynch* is similar to *ChSynch*; it is concerned with the new interactions in $dif\,(0.tr_t, tr_t)$, $dif\,(1.tr_t, tr_t)$, and $dif\,(tr'_t, tr_t)$. This means that, at the head of the trace in the first position of these timed traces, we have the first interaction that has been recorded since the start of the external choice.

$$TCM \widehat{=} \left(\begin{array}{l}(0.ok, 0.wait, dif\,(0.tr_t, tr_t), 0.state),\\ (1.ok, 1.wait, dif\,(1.tr_t, tr_t), 1.state)\end{array}\right) TChSynch\,(ok', wait', dif\,(tr'_t, tr_t), state')$$

The definition of the relation *TChSynch* is as shown below.

$$\left(\begin{array}{l}(0.ok, 0.wait, 0.tr_t, 0.state),\\ (1.ok, 1.wait, 1.tr_t, 1.state)\end{array}\right) TChSynch\,(ok', wait', tr'_t, state') \Leftrightarrow \left(\begin{array}{l}Diverge_t \vee WaitokNoEvent_t \vee\\ DoAnyEvent_t \vee Terminate_t\end{array}\right)$$

Like *Diverge*, the predicate $Diverge_t$ is used to capture the diverging behaviour of the timed choice.

$$Diverge_t \widehat{=} ok \wedge \neg ok' \wedge \left(\begin{array}{l}(0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr_t = tr'_t \wedge 0.state = state') \vee\\ (1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr_t = tr'_t \wedge 1.state = state')\end{array}\right)$$

Next we consider the waiting state of the external choice; the requirement $tr' = \langle\rangle$ in the definition of the standard

choice operator is now a restriction on $Flat(tr'_t)$ instead.

$$WaitokNoEvent_t \mathrel{\widehat{=}} \left(\begin{array}{l} ok' \wedge 0.ok = ok' \wedge 1.ok = ok' \wedge \\ wait' \wedge 0.wait = wait' \wedge 1.wait = wait' \wedge \\ Flat(tr'_t) = \langle\rangle \wedge 0.tr_t = tr'_t \wedge 1.tr_t = tr'_t \wedge \\ 0.state = state' \wedge 1.state = state' \end{array} \right)$$

Termination is considered below in the definition of the predicate $Terminate_t$.

$$Terminate_t \mathrel{\widehat{=}} ok \wedge \left(\begin{array}{l} \neg wait' \wedge Flat(tr'_t) = \langle\rangle \wedge \\ \left(\begin{array}{l} (0.ok = ok' \wedge 0.wait = wait' \wedge 0.tr_t = tr'_t \wedge 0.state = state') \vee \\ (1.ok = ok' \wedge 1.wait = wait' \wedge 1.tr_t = tr'_t \wedge 1.state = state') \end{array} \right) \end{array} \right)$$

The predicate $DoAnyEvent_t$ is similar to $DoAnyEvent$, but now we need to consider the timer events.

$$DoAnyEvent_t \mathrel{\widehat{=}} ok' \wedge Flat(tr'_t) \neq \langle\rangle \wedge \left(\begin{array}{l} DoTimePassing \vee \\ DoSetup \vee \\ DoAnyOut \vee DoEventWithOut \vee DoOrderedOut \vee \\ DoEvent \end{array} \right)$$

In the above definition, each of the predicates captures a particular situation. The predicate $DoTimePassing$ registers the passage of time if both actions register empty traces at the head of the timed traces.

$$DoTimePassing \mathrel{\widehat{=}} \left(\begin{array}{l} fst(head(tr'_t)) = \langle\rangle \wedge fst(head(0.tr_t)) = \langle\rangle \wedge fst(head(1.tr_t)) = \langle\rangle \wedge \\ \left(\begin{array}{l} (0.ok, 0.wait, tail(0.tr_t), 0.state), \\ (1.ok, 1.wait, tail(1.tr_t), 1.state) \end{array} \right) TChSynch\ (ok', wait', tail(tr'_t), state') \end{array} \right)$$

The predicate $DoSetup$ describes the behaviour of the program when a *setup* event is observed at the head of the resulting timed trace. Such a situation arises when one of the actions engages in the *setup* event, which is shown at the head of its timed trace. The choice is not solved, and after the *setup* event is removed, the timed traces of both actions are further considered in the choice.

$$DoSetup \mathrel{\widehat{=}} \exists i, d \bullet \left(\begin{array}{l} head(fst(head(tr'_t))) = setup.i.d \wedge \\ \left(\begin{array}{l} ((head(fst(head(0.tr_t))) = setup.i.d) \wedge StepFirst) \vee \\ ((head(fst(head(1.tr_t))) = setup.i.d) \wedge StepSecond) \end{array} \right) \end{array} \right)$$

The predicates $StepFirst$ and $StepSecond$ specify the merge of the variables, taking into account a modified timed trace of the action that engaged in the *setup* event. In the new trace, the *setup* event is eliminated, but all the other events are kept, and so the event that follows the *setup* can contribute to the choice resolution.

$$StepFirst \mathrel{\widehat{=}} \left(\begin{array}{l} \left(\begin{array}{l} (0.ok, 0.wait, \langle(tail(fst(head(0.tr_t))), snd(head(0.tr_t)))\rangle \frown tail(0.tr_t), 0.state), \\ (1.ok, 1.wait, 1.tr_t, 1.state) \end{array} \right) \\ \quad TChSynch \\ (ok', wait', \langle(tail(fst(head(tr'_t))), snd(head(tr'_t)))\rangle \frown tail(tr'_t), state') \end{array} \right)$$

While $StepFirst$ covers the case in which the first action can engage in a *setup* event, the similar predicate $StepSecond$ is concerned with the second action. If both actions can engage in a *setup* event, then the choice of which appears first in the trace of the external choice is nondeterministic.

$$StepSecond \mathrel{\widehat{=}} \left(\begin{array}{l} \left(\begin{array}{l} (0.ok, 0.wait, 0.tr_t, 0.state), \\ (1.ok, 1.wait, \langle(tail(fst(head(1.tr_t))), snd(head(1.tr_t)))\rangle \frown tail(1.tr_t), 1.state) \end{array} \right) \\ \quad TChSynch \\ (ok', wait', \langle(tail(fst(head(tr'_t))), snd(head(tr'_t)))\rangle \frown tail(tr'_t), state') \end{array} \right)$$

The predicate $DoAnyOut$ captures the behaviour when one, but not both, of the actions can perform an *out*

event: the choice is not made, but the *out* event is observed in the trace of the external choice.

$$DoAnyOut \mathrel{\widehat{=}} \exists\, i, d \bullet \left(\begin{array}{l} head(fst(head(tr'_t))) = out.i.d \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} head(fst(head(0.tr_t))) = out.i.d \wedge \\ \forall\, j, n \bullet i \neq j \wedge head(fst(head(1.tr_t))) \neq out.j.n \wedge \\ StepFirst \end{array} \right) \vee \\ \left(\begin{array}{l} head(fst(head(1.tr_t))) = out.i.d \wedge \\ \forall\, j, n \bullet i \neq j \wedge head(fst(head(0.tr_t))) \neq out.j.n \wedge \\ StepSecond \end{array} \right) \end{array} \right) \end{array} \right)$$

The next predicate considers the situation in which both actions can perform the event *out*; in this case, the external choice is still not solved, but the choice of which *out* event is observed in the trace of the external choice first is based on the delay of the timers: the *out* event with the smallest delay is chosen. If the delays are the same, then the order of the events is nondeterministic.

$$DoOrderedOut \mathrel{\widehat{=}} \exists\, i, d \bullet \left(\begin{array}{l} head(fst(head(tr'_t))) = out.i.d \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} head(fst(head(0.tr_t))) = out.i.d \wedge \\ \exists\, j, n \bullet (head(fst(head(1.tr_t))) = out.j.n) \wedge i \neq j \wedge \\ StepFirst \wedge (d \leq n) \end{array} \right) \vee \\ \left(\begin{array}{l} head(fst(head(1.tr_t))) = out.i.d \wedge \\ \exists\, j, n \bullet (head(fst(head(0.tr_t))) = out.j.n) \wedge i \neq j \wedge \\ StepSecond \wedge (d \leq n) \end{array} \right) \end{array} \right) \end{array} \right)$$

The predicate *DoEvent* is concerned with the standard events, that is, those that are not timer events. It is very similar to the *DoAnyEvent* predicate used in the definition of the standard external choice.

$$DoEvent \mathrel{\widehat{=}} \exists\, a \bullet \left(\begin{array}{l} head(fst(head(tr'_t))) = a \wedge a \notin timerEvents \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} head(fst(head(0.tr_t))) = a \wedge \\ head(fst(head(1.tr_t))) \notin timerEvents \wedge \\ (ok' = 0.ok \wedge wait' = 0.wait \wedge tr'_t = 0.tr_t \wedge state' = 0.state) \end{array} \right) \vee \\ \left(\begin{array}{l} head(fst(head(1.tr_t))) = a \wedge \\ head(fst(head(0.tr_t))) \notin timerEvents \wedge \\ (ok' = 1.ok \wedge wait' = 1.wait \wedge tr'_t = 1.tr_t \wedge state' = 1.state) \end{array} \right) \end{array} \right) \end{array} \right)$$

As defined in Sect. 4.1, the set *timerEvents* contains the relevant *setup* and *out* events.

Finally, *DoEventWithOut* specifies the situation in which an event *out* is possible in the presence of an ordinary event. The choice offers both of them: as described above by *DoAnyOut*, if the *out* event is selected, then the choice is not solved and the other event can occur. If the other event is selected, then the event *halt*, related to the same timer as the *out* event, is observed in the trace immediately afterwards.

$$DoEventWithOut \mathrel{\widehat{=}} \exists\, a \bullet \left(\begin{array}{l} head(fst(head(tr'_t))) = a \wedge a \notin timerEvents \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} head(fst(head(0.tr_t))) = a \wedge \\ \exists\, i, d \bullet head(fst(head(1.tr_t))) = out.i.d \wedge i \neq j \wedge \\ \left(\begin{array}{l} ok' = 0.ok \wedge wait' = 0.wait \wedge state' = 0.state \wedge \\ tr'_t = addHead(a, i, 0.tr_t) \frown tail(0.tr_t) \end{array} \right) \end{array} \right) \vee \\ \left(\begin{array}{l} head(fst(head(1.tr_t))) = a \wedge \\ \exists\, i, d \bullet head(fst(head(0.tr_t))) = out.i.d \wedge i \neq j) \wedge \\ \left(\begin{array}{l} ok' = 1.ok \wedge wait' = 1.wait \wedge state' = 1.state \wedge \\ tr'_t = addHead(a, i, 1.tr_t) \frown tail(1.tr_t) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)$$

The sequence $addHead(a, i, t) \mathrel{\widehat{=}} \langle (\langle a, halt.i \rangle \frown tail(fst(head(t))), snd(head(t))) \rangle$ is a singleton. The validation of the definition of this special external choice operator is provided by the profs of many laws that involve this operator [She06], a few of which are presented below.

Our new choice operator is idempotent, commutative, and associative. In addition, the following laws, where we take c_i to be ordinary events, as opposed to timer events, hold; they confirm the special nature of a choice involving timer events. First, a *setup* event is not determinant in the choice and does not affect it. When in choice with other events, ordinary or timer events, a *setup* event has priority.

Law 6 $(setup.i.d \rightarrow A) \boxtimes B = setup.i.d \rightarrow (A \boxtimes B)$ **provided** $\neg \exists j, n \bullet setup.j.n \in initials(B)$.

If two setup events are available for choice, then either of them may be chosen.

Law 7

$$\begin{array}{l}(setup.i.d \rightarrow A) \boxtimes (setup.j.n \rightarrow B)\\ =\\ (setup.i.d \rightarrow (A \boxtimes (setup.j.n \rightarrow B))) \sqcap (setup.j.n \rightarrow (setup.i.d \rightarrow A \boxtimes B))\end{array}$$

As in the case of *setup*, an *out* event is not determinant in the choice, but differently from *setup*, the *out* event offers the ordinary events c_i as a choice using the standard external choice operator.

Law 8

$$\begin{array}{l}(out.j.d \rightarrow A) \boxtimes (\Box\, i : 1 \mathinner{..} n \bullet c_i \rightarrow B_i)\\ =\\ (out.j.d \rightarrow (A \boxtimes (\Box\, i : 1 \mathinner{..} n \bullet c_i \rightarrow B_i))) \Box (\Box\, i : 1 \mathinner{..} n \bullet c_i \rightarrow halt.j \rightarrow B_i)\end{array}$$

In the case where the choice operator $\boxtimes$ involves two *out* events, the next law states that they are offered in the order determined by the delays: the event with the shorter delay is offered first.

Law 9

$$(out.i.d \rightarrow A) \boxtimes (out.j.n \rightarrow B) = \begin{cases} out.i.d \rightarrow (A \boxtimes (out.j.n \rightarrow B)), & \text{if } d > n;\\ out.j.n \rightarrow ((out.i.d \rightarrow A) \boxtimes B), & \text{if } d < n;\\ (out.i.d \rightarrow (A \boxtimes (out.j.n \rightarrow B))) \sqcap (out.j.n \rightarrow (out.i.d \rightarrow A \boxtimes B)), & \\ \quad \text{if } d = n. & \end{cases}$$

provided $i \neq j$.

The next law states that the choice operator $\boxtimes$ is solved between timer events.

Law 10 $((out.i.d \rightarrow A) \Box C) \boxtimes ((out.j.n \rightarrow B) \Box D) = ((out.i.d \rightarrow A) \boxtimes (out.j.n \rightarrow B)) \Box (C \Box D)$

The final law for the operator $\boxtimes$ states that if all actions in the choice do not start with timer events, then the choice is the same as that characterised by the standard external choice operator.

Law 11 $A \boxtimes B = A \Box B$
provided $timerEvents \cap (initials(A) \cup initials(B)) \neq \emptyset$.

The set $initials(A)$ contains the events in which the action A is prepared to engage. Its definition for *Circus* can be found in [Fre06]. Together, these laws establish that an action $A \boxtimes B$ can always be reduced to an action $A' \Box B'$ such that A' and B' do not contain the operator $\boxtimes$.

4.2.2. Normal form parallel composition

We also need to introduce a new parallel operator $A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} B$, and some expansion laws, that capture the special nature of the synchronisation and interleaving of timer and ordinary events. The semantics of this new operator is given using the parallel by merge operator of the UTP.

$$A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} B \mathrel{\widehat{=}} A \|_{NPM} B$$

In this case, the merge predicate is NPM, which is defined as shown below.

$$NPM(cs, s_A, s_B) \mathrel{\widehat{=}} \left(\begin{array}{l} ok' = (0.ok \wedge 1.ok) \wedge \\ wait' = (0.wait \vee 1.wait) \wedge \\ state' = (s_B \lhd 0.state) \cup (s_A \lhd 1.state) \wedge \\ dif(tr'_t, tr_t)\ NTSync(cs)\ dif(0.tr, tr), dif(1.tr, tr) \end{array} \right)$$

The synchronisation relation $NTSync$ associates prefixed traces $0.tr_t$ and $1.tr_t$ of the parallel actions, a synchronisation set cs, and the combined trace tr_t of the parallelism. It is concerned with the interactions

$dif(0.tr, tr)$, $dif(1.tr, tr)$, $dif(tr'_t, tr_t)$ that are recorded after the parallelism starts.

$$tr_t\ NTSync(cs)\ 0.tr_t, 1.tr_t \mathrel{\widehat{=}} \left(\begin{array}{l} (0.tr_t = \langle\rangle \wedge 1.tr_t = \langle\rangle \wedge tr_t = \langle\rangle) \vee \\ (0.tr_t \neq \langle\rangle \wedge 1.tr_t = \langle\rangle \wedge tr_t\ NTSync(cs)\ 0.tr_t, \langle(\langle\rangle, \{\})\rangle) \vee \\ (0.tr_t = \langle\rangle \wedge 1.tr_t \neq \langle\rangle \wedge tr_t\ NTSync(cs)\ \langle(\langle\rangle, \{\})\rangle, 1.tr_t) \vee \\ \exists\, t_1, r_1, t_2, r_2 \bullet \left(\begin{array}{l} (0.tr_t = \langle(t_1, r_1)\rangle \frown S_1) \wedge (1.tr_t = \langle(t_2, r_2)\rangle \frown S_2) \wedge \\ fst(head(tr_t))\ NPSync(cs)\ t_1, t_2 \wedge \\ snd(head(tr_t)) = ((r_1 \cup r_2) \cap cs) \cup ((r_1 \cap r_2) \backslash cs) \\ tail(tr_t) NTSync(cs)\ S_1, S_2 \end{array}\right) \end{array}\right)$$

NTSync is defined as a disjunction. The first disjunct handles the case in which the prefixed traces are empty: the combined trace is also empty. Next we have the case in which just one of the prefixed timed traces is empty: the synchronisation is carried out between the non-empty trace and $\langle(\langle\rangle, \{\})\rangle$. Finally, if both prefixed traces are not empty, then we synchronise the first time slots of the two traces: we synchronise the trace element of each timed trace using a new relation *NPSync* defined below.

$$tr\ NPSync(cs)\ 0.tr, 1.tr \mathrel{\widehat{=}} EmptyTraces \vee SynchEvents \vee NoSynchEvents \vee doSetupEvent \vee doOutEvent$$

The predicate *EmptyTraces* considers the case in which both prefixed traces are empty, in which case the combined trace is empty as well. If any of the two traces is not empty, then the *Circus* untimed synchronisation relation *Sync* [She06, Oli06] is used; it is similar to the standard CSP synchronisation relation [Ros98].

$$EmptyTraces \mathrel{\widehat{=}} \left(\begin{array}{l} (0.tr = \langle\rangle \wedge 1.tr = \langle\rangle \wedge tr = \langle\rangle) \vee \\ (0.tr \neq \langle\rangle \wedge 1.tr = \langle\rangle \wedge tr \in Sync(0.tr, 1.tr, cs)) \vee \\ (0.tr = \langle\rangle \wedge 1.tr \neq \langle\rangle \wedge tr \in Sync(0.tr, 1.tr, cs)) \end{array}\right)$$

SynchEvents describes the situation in which the event observed at the heads of the traces is in the synchronisation set *cs*. If the event at the head of the combined trace is in *cs*, then the prefixed traces should both have the same event at their heads, and the tails of all traces should also be related by $NPSync(cs)$.

$$SynchEvents \mathrel{\widehat{=}} \left(\begin{array}{l} (head(tr) \in cs) \wedge (head(0.tr) = head(tr)) \wedge (head(1.tr) = head(tr)) \wedge \\ tail(tr)\ NPSync(cs)\ tail(0.tr), tail(1.tr) \end{array}\right)$$

NoSynchEvents describes the behaviour when the element at the head of the combined trace is not a member of *cs* and is not a timer event. This is only possible if the head event of $0.tr$ (or $1.tr$) is that same event, and the head event of $1.tr$ ($0.tr$) is not a *setup*. This gives priority to the *setup* event in the parallelism.

$$NoSynchEvents \mathrel{\widehat{=}} \left(\begin{array}{l} (head(tr) \notin cs) \wedge (head(tr) \notin timerEvents) \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} (head(0.tr) = head(tr)) \wedge \\ \forall\, i, d \bullet head(1.tr) \neq setup.i.d \wedge \\ tail(tr)\ NPSync(cs)\ tail(0.tr), 1.tr \end{array}\right) \vee \\ \left(\begin{array}{l} (head(1.tr) = head(tr)) \wedge \\ \forall\, i, d \bullet head(0.tr) \neq setup.i.d \wedge \\ tail(tr)\ NPSync(cs)\ 0.tr, tail(1.tr) \end{array}\right) \end{array}\right) \end{array}\right)$$

The next predicate, *doSetupEvent*, deals in particular with the situation in which a *setup* event is observed at the head of the combined trace. It must be a head event of one of the prefixed traces.

$$doSetupEvent \mathrel{\widehat{=}} \exists\, i, d \bullet head(tr) = setup.i.d \wedge \left(\begin{array}{l} \left(\begin{array}{l} head(0.tr) = head(tr) \wedge \\ (tail(tr)\ NPSync(cs)\ tail(0.tr), 1.tr) \end{array}\right) \vee \\ \left(\begin{array}{l} head(1.tr) = head(tr) \wedge \\ (tail(tr)\ NPSync(cs)\ 0.tr, tail(1.tr)) \end{array}\right) \end{array}\right)$$

Finally, *doOutEvent* is similar to *doSetupEvent*, except that it considers *out* events, which are ordered by their delays. If an *out* event is observed at the head of the combined trace, then it must be at the head of $0.tr$ or $1.tr$.

If an *out* event is at the top of both of them, then that with the smallest delay is chosen.

$$doOutEvent \mathrel{\widehat{=}} \exists\, i, d \bullet \left(\begin{array}{l} head(tr) = out.i.d \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} head(0.tr) = head(tr) \wedge \\ \left(\begin{array}{l} (\forall j, n \bullet (head(1.tr \downarrow timerEvents) \neq out.j.n) \wedge i \neq j) \vee \\ (\exists j, n \bullet head(1.tr \downarrow timerEvents) = out.j.n \wedge i \neq j) \Rightarrow d \leq n \end{array} \right) \wedge \\ (tail(tr)\ NPSync(cs)\ tail(0.tr), 1.tr) \end{array} \right) \\ \vee \\ \left(\begin{array}{l} head(1.tr) = head(tr) \wedge \\ \left(\begin{array}{l} (\forall j, n \bullet (head(0.tr \downarrow timerEvents) \neq out.j.n) \wedge i \neq j) \vee \\ (\exists j, n \bullet head(0.tr \downarrow timerEvents) = out.j.n \wedge i \neq j) \Rightarrow d \leq n \end{array} \right) \\ (tail(tr)\ NPSync(cs)\ 0.tr, tail(1.tr)) \end{array} \right) \end{array} \right) \end{array} \right)$$

We present below step laws that can be used to convert a parallel action into a sequential action. They give an algebraic semantics for the new operator, and confirm the behaviour associated with timer events. We omit laws that involve only ordinary events; they are standard and also valid for the CSP parallel operator [Ros98]. The first law states that *setup* events, when in parallel, have to occur before any other events.

Law 12

$$\begin{array}{l} (setup.i.d \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} (setup.j.e \rightarrow B) \\ = \\ ((setup.i.d \rightarrow setup.j.e \rightarrow Skip) \sqcap (setup.j.e \rightarrow setup.i.d \rightarrow Skip));\ (A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} B) \end{array}$$

provided $i \neq j$.

The next two laws state the order in which *out* events occur based on their delays. If both parallel actions are ready to engage in an *out* event, but they have different delays, that with the smallest delay occurs first.

Law 13

$$((out.i.d \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} (out.j.n \rightarrow B)) = out.i.d \rightarrow (A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} (out.j.n \rightarrow B))$$

provided $d < n$.

If the delays are the same, then the order in which the *out* events occur is nondeterministic.

Law 14

$$\begin{array}{l} (out.i.d \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} (out.j.n \rightarrow B) \\ = \\ ((out.i.d \rightarrow out.j.n \rightarrow Skip) \sqcap (out.j.n \rightarrow out.i.d \rightarrow Skip));\ (A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} B) \end{array}$$

provided $d = n$.

The following two laws capture the behaviour when an *out* event is offered in parallel with ordinary events: they do not have priority, and are guaranteed to occur first only if the ordinary events are all in the synchronisation set, and therefore, blocked by the parallelism.

Law 15

$$\begin{array}{l} ((out.i.d \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} (\Box\, j : 1 \mathinner{..} n \bullet c_j \rightarrow B_j) \\ = \\ \left(\begin{array}{l} out.i.d \rightarrow (A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} (\Box\, j : 1 \mathinner{..} n \bullet c_j \rightarrow B_j)) \\ \sqcap \\ (\Box\, j : 1 \mathinner{..} n \bullet c_j \rightarrow ((out.i.d \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} B_j)) \end{array} \right) \end{array}$$

provided for all j, $(c_j \notin cs) \wedge (c_j \neq out.i.d)$, for any timer index i and delay d.

Law 16

$$((out.i.d \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} (\Box\, j : 1 \mathinner{..} n \bullet c_j \rightarrow B_j))$$
$$=$$
$$(out.i.d \rightarrow (A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} (\Box\, j : 1 \mathinner{..} n \bullet c_j \rightarrow B_j)))$$

provided for all j, $(c_j \in cs) \wedge (c_j \neq out.i.d)$ for any timer index i and delay d.

The next laws consider the case in which the parallel actions can either do *out* or ordinary events. In this case, the *out* event with the shorter delay has priority.

Law 17

$$((out.i.d \rightarrow A) \Box B) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} ((out.j.n \rightarrow C) \Box D)$$
$$=$$
$$out.i.d \rightarrow (A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} ((out.j.n \rightarrow C) \Box D))$$

provided $d < n$.

If the delays in the parallel *out* events are the same, then either can be chosen.

Law 18

$$((out.i.d \rightarrow A) \Box B) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} ((out.j.n \rightarrow C) \Box D)$$
$$=$$
$$\left(\begin{array}{l} (out.i.d \rightarrow (A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} ((out.j.n \rightarrow C) \Box D))) \\ \Box \\ (out.j.n \rightarrow (((out.i.d \rightarrow A) \Box B) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} C)) \end{array} \right)$$

provided $d = n$.

Together, these laws (and the omitted standard step laws) establish that any action $A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} B$ can be reduced to an action $A' \Box B'$ where A' and B' do not contain the normal form operators.

Finally, we present a law that relates the new and the original parallel operators.

Law 19 $A \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} B = A \llbracket s_A \mid cs \mid s_B \rrbracket B$
provided $\{setup, out, halt\} \cap (usedC(A, B)) = \emptyset$.

The set $usedC(A, B)$ contains all the channels used by the actions A and B [Oli06]. This law establishes that it is not always necessary to sequentialise the normal form parallelism to eliminate it. If the actions in parallel have no timer events, then the normal form parallelism reduces to the standard parallelism. The definition of the normalisation function Φ, which uses the new operators, is presented in the next section.

4.3. Reducing timed actions to actions with timer events

The action $\Phi(A)$ uses the timer events, instead of time operators, to specify the same time constraints as A. The function Φ is an identity for the basic actions (except for *Wait*), and distributes through the binary operators, except for timeout, external choice and parallel composition. For *Wait* d, Φ gives an action that initialises a timer using the event $setup.i.d$, and then waits for the occurrence of $out.i.d$.

$$\Phi(Wait\ d) = (setup.i.d \rightarrow out.i.d \rightarrow Skip)$$

For the timeout operator, without loss of generality, we consider a particular case: $(\Box\, j : 1 \mathinner{..} n \bullet c_j \rightarrow A_j) \overset{d}{\rhd} B$, where the events c_j are ordinary. We consider the application of the timeout operator only to this special form of external choice because the timeout has to stop the corresponding timer (using the event $halt.i$) after the occurrence of the first event. This does not lead to loss of generality of our definition because, using the algebraic

laws of *Circus Time*, we can transform any action into the required form [She06].

$$\Phi((\Box\, j : 1 \mathinner{..} n \bullet c_j \rightarrow A_j) \overset{d}{\rhd} B) =$$
$$setup.i.d \rightarrow ((\Box\, j : 1 \mathinner{..} n \bullet c_j \rightarrow halt.i \rightarrow \Phi(A_j)) \Box (out.i.d \rightarrow \Phi(B)))$$

Like with *Wait d*, the action that corresponds to the timeout initialises a timer using $setup.i.d$. Afterwards, it offers a choice of synchronising on an event c_j or recording a timeout with the event $out.i.d$. If an event c_j occurs before d time units, then the timer is stopped with the event $halt.i$.

External choice is mapped by Φ to the new external choice operator $\boxtimes$ introduced in the previous section.

$$\Phi(A \Box B) = \Phi(A) \boxtimes \Phi(B)$$

Similarly a parallel composition is mapped by Φ to the new parallel operator.

$$\Phi(A \llbracket s_A \mid cs \mid s_B \rrbracket B) = \Phi(A) \llbracket s_A \mid cs \mid s_B \rrbracket^{nf} \Phi(B)$$

In summary, applying the definition of Φ, and the laws of the new choice and parallelism operators presented in the previous section, we can reduce any timed action to an action with timer events and no time operators. The soundness of this process is the subject of the next section.

4.4. Normalisation soundness

For every timed action A, its normalisation involves the introduction of a set of timers, $Timers(k, n)$, where k and n depend on the number $m = n - k + 1$ of such time operators in A, and the evaluation of the application of the Φ function. This is a purely syntactic process that generates a timer for each wait or timeout operator in A; the theorem below establishes its soundness.

Theorem 3 For any timed action A whose recursions do not involve any timed operators, there is an action $Timers(k, n)$ such that $A = \Phi(A)$ **par** $Timers(k, n)$, where $k..n$ is the integer range used to identify the timers, one for each wait or timeout action in the original A.

Proof. By structural induction on the *Circus Time* constructs; we include here only proofs for the time operators, which are the most affected by the normalisation.

Case *Wait d*: A single *Wait d* requires only one timer, so we show that $Wait\ d = \Phi(Wait\ d)$ **par** $Timer(i, d)$, where i is the arbitrary identifier of the timer used for this particular command.

$$\Phi(Wait\ d)\ \mathbf{par}\ Timer(i, d) \qquad \text{[definition of } \Phi\text{]}$$
$$= (setup.k.d \rightarrow out.k.d \rightarrow Skip)\ \mathbf{par}\ Timer(i, d) \qquad \text{[definition of } \mathbf{par}\text{]}$$
$$= (((setup.k.d \rightarrow out.k.d \rightarrow Skip);\ Terminate(i, i)) \llbracket \{\} \mid TSet \mid \{\} \rrbracket Timer(i, d)) \setminus TSet$$
$$\text{[definition of } Timer(i, d)\text{]}$$
$$= \left(\begin{array}{l} ((setup.k.d \rightarrow out.k.d \rightarrow Skip);\ Terminate(i, i)) \\ \quad \llbracket \{\} \mid TSet \mid \{\} \rrbracket \\ \left(\begin{array}{l} \mu X \bullet \left(\begin{array}{l} setup.k.d \rightarrow \left(\begin{array}{l} (halt.k \rightarrow X) \\ \Box\, (Wait\ d;\ ((out.k.d \rightarrow X) \Box (terminate.k \rightarrow Skip)) \\ \Box\, (terminate.k \rightarrow Skip) \end{array} \right) \\ \Box\, (terminate.k \rightarrow Skip) \end{array} \right) \end{array} \right) \end{array} \right) \setminus TSet$$
$$\text{[fixed point and definition of } Timer(i, d)\text{]}$$
$$= \left(\begin{array}{l} ((setup.k.d \rightarrow out.k.d \rightarrow Skip);\ Terminate(i, i)) \\ \quad \llbracket \{\} \mid TSet \mid \{\} \rrbracket \\ \left(\begin{array}{l} setup.k.d \rightarrow \left(\begin{array}{l} (halt.k \rightarrow Timer(k, d)) \\ \Box\, (Wait\ d;\ ((out.k.d \rightarrow Timer(k, d)) \Box (terminate.k \rightarrow Skip))) \\ \Box\, (terminate.k \rightarrow Skip) \end{array} \right) \\ \Box\, (terminate.k \rightarrow Skip) \end{array} \right) \end{array} \right) \setminus TSet$$
$$[setup.k.d \in TSet \text{ and step law (Law 21)}]$$

$$= \left(setup.k.d \rightarrow \left(\begin{array}{l} ((out.k.d \rightarrow Skip);\ Terminate(i,i)) \\ \quad [\![\{\} \mid TSet \mid \{\}]\!] \\ \left(\begin{array}{l} (halt.k \rightarrow Timer(k,d)) \\ \Box\ (Wait\ d;\ ((out.k.d \rightarrow Timer(i,d)) \Box (terminate.k \rightarrow Skip))) \\ \Box\ (terminate.k \rightarrow Skip) \end{array} \right) \end{array} \right) \right) \setminus TSet$$

[$\{ out.k.d, halt.k, terminate.k \} \subseteq TSet$ and step law (Law 20)]

$$= \left(setup.k.d \rightarrow Wait\ d;\ \left(\begin{array}{l} ((out.k.d \rightarrow Skip);\ Terminate(i,i)) \\ \quad [\![\{\} \mid TSet \mid \{\}]\!] \\ ((out.k.d \rightarrow Timer(i,d)) \Box (terminate.k \rightarrow Skip)) \end{array} \right) \right) \setminus TSet$$

[$out.k.d \in TSet$ and step law (Law 21)]

$$= (setup.k.d \rightarrow Wait\ d;\ out.k.d \rightarrow Skip;\ (Terminate(i,i) [\![\{\} \mid TSet \mid \{\}]\!] Timer(i,d))) \setminus TSet$$

[property of hiding (Law 30)]

$$= (setup.k.d \rightarrow Wait\ d;\ out.k.d \rightarrow Skip) \setminus TSet;\ (Terminate(i,i) [\![\{\} \mid TSet \mid \{\}]\!] Timer(i,d)) \setminus TSet$$

[Lemma 1]

$$= ((setup.k.d \rightarrow Wait\ d;\ out.k.d \rightarrow Skip) \setminus TSet);\ Skip$$ [right unit of sequence]

$$= (setup.k.d \rightarrow Wait\ d;\ out.k.d \rightarrow Skip) \setminus TSet$$ [properties of hiding (Law 25)]

$$= (Wait\ d;\ out.k.d \rightarrow Skip) \setminus TSet$$ [property of hiding (Law 30)]

$$= (Wait\ d \setminus TSet);\ ((out.k.d \rightarrow Skip) \setminus TSet)$$ [properties of hiding (Laws 27 and 25)]

$$= Wait\ d;\ Skip$$ [right unit of sequence]

$$= Wait\ d$$ □

Case $(c \rightarrow A) \overset{d}{\triangleright} B$**:** The induction hypothesis allows us to assume that $\Phi(A)$ **par** $Timers(k,n) = A$ and $\Phi(B)$ **par** $Timers(j,m) = B$ with disjoint timer index intervals (k,n) and (j,m). What we need to prove is that $\Phi((c \rightarrow A) \overset{d}{\triangleright} B)$ in parallel with an interleaving of the timers $Timers(k,n)$ for A and $Timers(j,m)$ for B, and an additional timer $Timer(i,d)$ for the timeout operator, with a fresh index i, has the same behaviour as the timeout action $(c \rightarrow A) \overset{d}{\triangleright} B$. We proceed as shown below.

$$\Phi((c \rightarrow A) \overset{d}{\triangleright} B)\ \mathbf{par}\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m))$$ [definition of Φ]

$$= \left(\begin{array}{l} setup.i.d \rightarrow ((c \rightarrow halt.k \rightarrow \Phi(A)) \Box (out.k.d \rightarrow \Phi(B))) \\ \quad \mathbf{par} \\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m)) \end{array} \right)$$

[int is fresh and properties of hiding (Laws 28, 25, 26, and 31)]

$$= \left(\begin{array}{l} setup.i.d \rightarrow ((c \rightarrow halt.k \rightarrow \Phi(A)) \Box (out.k.d \rightarrow int \rightarrow \Phi(B))) \setminus \{int\}) \\ \quad \mathbf{par} \\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m)) \end{array} \right)$$

[definition of **par** (s_{AB} is the set of variables that can be changed by A and B]

$$= \left(\begin{array}{l} \left(\begin{array}{l} setup.i.d \rightarrow ((c \rightarrow halt.k \rightarrow \Phi(A)) \Box (out.k.d \rightarrow int \rightarrow \Phi(B))) \setminus \{int\}; \\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m)) \end{array} \right) \\ \quad [\![s_{AB} \mid TSet \mid \{\}]\!] \\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m)) \end{array} \right) \setminus TSet$$

[int is fresh and properties of hiding (Laws 26, 28, 30, 32, and 29)]

$$= \left(\begin{array}{l} \left(\begin{array}{l} setup.i.d \rightarrow ((c \rightarrow halt.k \rightarrow \Phi(A)) \,\Box\, (out.k.d \rightarrow int \rightarrow \Phi(B))); \\ (Terminate(i,i) \,|||\, Terminate(k,n) \,|||\, Terminate(j,m)) \end{array} \right) \\ [\![s_{AB} \mid TSet \mid \{\}]\!] \\ (Timer(i,d) \,|||\, Timers(k,n) \,|||\, Timers(j,m)) \end{array} \right) \setminus TSet \cup \{int\}$$

[definition of $Timer(i,d)$ and fixed point]

$$= \left(\begin{array}{l} \left(\begin{array}{l} setup.i.d \rightarrow ((c \rightarrow halt.k \rightarrow \Phi(A)) \,\Box\, (out.k.d \rightarrow int \rightarrow \Phi(B))); \\ (Terminate(i,i) \,|||\, Terminate(k,n) \,|||\, Terminate(j,m)) \end{array} \right) \\ [\![s_{AB} \mid TSet \mid \{\}]\!] \\ \left(\begin{array}{l} \left(\begin{array}{l} setup.i.d \rightarrow \left(\begin{array}{l} (halt.k \rightarrow Timer(i,d)) \\ \Box\, Wait\ d; \left(\begin{array}{l} (out.k.d \rightarrow Timer(i,d)) \\ \Box \\ (terminate.k \rightarrow Skip) \end{array} \right) \\ \Box\, (terminate.i \rightarrow Skip) \end{array} \right) \\ \Box\, (terminate.i \rightarrow Skip) \end{array} \right) \,|||\, \\ Timers(k,n) \,|||\, Timers(j,m) \end{array} \right) \end{array} \right) \setminus TSet \cup \{int\}$$

[step law (Law 24)]

$$= \left(setup.i.d \rightarrow \left(\begin{array}{l} \left(\begin{array}{l} ((c \rightarrow halt.k \rightarrow \Phi(A)) \,\Box\, (out.k.d \rightarrow int \rightarrow \Phi(B))); \\ (Terminate(i,i) \,|||\, Terminate(k,n) \,|||\, Terminate(j,m)) \end{array} \right) \\ [\![s_{AB} \mid TSet \mid \{\}]\!] \\ \left(\begin{array}{l} \left(\begin{array}{l} (halt.k \rightarrow Timer(i,d)) \\ \Box\, Wait\ d; \left(\begin{array}{l} (out.k.d \rightarrow Timer(i,d)) \\ \Box \\ (terminate.k \rightarrow Skip) \end{array} \right) \\ \Box\, (terminate.i \rightarrow Skip) \end{array} \right) \,|||\, \\ Timers(k,n) \,|||\, Timers(j,m) \end{array} \right) \end{array} \right) \right) \setminus TSet \cup \{int\}$$

[$setup.i.d \in TSet$ and properties of hiding (Laws 29 and 25)]

$$= \left(\begin{array}{l} \left(\begin{array}{l} ((c \rightarrow halt.k \rightarrow \Phi(A)) \,\Box\, (out.k.d \rightarrow int \rightarrow \Phi(B))); \\ (Terminate(i,i) \,|||\, Terminate(k,n) \,|||\, Terminate(j,m)) \end{array} \right) \\ [\![s_{AB} \mid TSet \mid \{\}]\!] \\ \left(\begin{array}{l} \left(\begin{array}{l} (halt.k \rightarrow Timer(i,d)) \\ \Box\, Wait\ d; \left(\begin{array}{l} (out.k.d \rightarrow Timer(i,d)) \\ \Box \\ (terminate.k \rightarrow Skip) \end{array} \right) \\ \Box\, (terminate.i \rightarrow Skip) \end{array} \right) \,|||\, \\ Timers(k,n) \,|||\, Timers(j,m) \end{array} \right) \end{array} \right) \setminus TSet \cup \{int\}$$

[distribution of sequence and parallelism over choice]

$$
= \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} (c \rightarrow halt.k \rightarrow \Phi(A)); \\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m)) \end{array} \right) \\
[\![s_{AB} \mid TSet \mid \{\}]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l} (halt.k \rightarrow Timer(i,d)) \\ \Box\ Wait\ d;\ \left(\begin{array}{l} (out.k.d \rightarrow Timer(i,d)) \\ \Box \\ (terminate.k \rightarrow Skip) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \mid\mid\mid \\
Timers(k,n) \mid\mid\mid Timers(j,m)
\end{array} \right)
\end{array} \right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l} (out.k.d \rightarrow int \rightarrow \Phi(B)); \\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m)) \end{array} \right) \\
[\![s_{AB} \mid TSet \mid \{\}]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l} (halt.k \rightarrow Timer(i,d)) \\ \Box\ Wait\ d;\ \left(\begin{array}{l} (out.k.d \rightarrow Timer(i,d)) \\ \Box \\ (terminate.k \rightarrow Skip) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \mid\mid\mid \\
Timers(k,n) \mid\mid\mid Timers(j,m)
\end{array} \right)
\end{array} \right)
\end{array} \right) \setminus TSet \cup \{int\}
$$

[step laws (Laws 23 and 21)]

$$
= \left(\begin{array}{l}
\left(\begin{array}{l}
c \rightarrow halt.k \rightarrow \\
\left(\begin{array}{l} (\Phi(A);\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m))) \\ \quad [\![s_{AB} \mid TSet \mid \{\}]\!] \\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m)) \end{array} \right)
\end{array} \right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l} (out.k.d \rightarrow int \rightarrow \Phi(B)); \\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m)) \end{array} \right) \\
[\![s_{AB} \mid TSet \mid \{\}]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l} (halt.k \rightarrow Timer(i,d)) \\ \Box\ Wait\ d;\ \left(\begin{array}{l} (out.k.d \rightarrow Timer(i,d)) \\ \Box \\ (terminate.k \rightarrow Skip) \end{array} \right) \\ \Box\ (terminate.i \rightarrow Skip) \end{array} \right) \mid\mid\mid \\
Timers(k,n) \mid\mid\mid Timers(j,m)
\end{array} \right)
\end{array} \right)
\end{array} \right) \setminus TSet \cup \{int\}
$$

[step laws (Laws 20, 21 and 23]

$$
= \left(\begin{array}{l}
\left(\begin{array}{l}
c \rightarrow halt.k \rightarrow \\
\left(\begin{array}{l} (\Phi(A);\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m))) \\ \quad [\![s_{AB} \mid TSet \mid \{\}]\!] \\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m)) \end{array} \right)
\end{array} \right) \\
\Box \\
\left(\begin{array}{l}
Wait\ d;\ out.k.d \rightarrow int \rightarrow \\
\left(\begin{array}{l} (\Phi(B);\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m))) \\ \quad [\![s_{AB} \mid TSet \mid \{\}]\!] \\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m)) \end{array} \right)
\end{array} \right)
\end{array} \right) \setminus TSet \cup \{int\}
$$

[properties of hiding (Laws 29, 31, 25, 26, 30, and 28)]

$$= \left(\begin{array}{l} \left(\begin{array}{l} c \rightarrow \\ \left(\begin{array}{l} (\Phi(A);\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m))) \\ \quad [\![s_{AB} \mid TSet \mid \{\}]\!] \\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m)) \end{array} \right) \setminus TSet \end{array} \right) \\ \Box \\ \left(\begin{array}{l} Wait\ d;\ int \rightarrow \\ \left(\begin{array}{l} (\Phi(B);\ (Terminate(i,i) \mid\mid\mid Terminate(k,n) \mid\mid\mid Terminate(j,m))) \\ \quad [\![s_{AB} \mid TSet \mid \{\}]\!] \\ (Timer(i,d) \mid\mid\mid Timers(k,n) \mid\mid\mid Timers(j,m)) \end{array} \right) \setminus TSet \end{array} \right) \end{array} \right) \setminus \{int\}$$

[step law (Law 24) and Lemma 1]

$$= \left(\begin{array}{l} (c \rightarrow ((\Phi(A);\ Terminate(k,n)) [\![s_{AB} \mid TSet \mid \{\}]\!] Timers(k,n)) \setminus TSet \\ \Box \\ (Wait\ d;\ int \rightarrow ((\Phi(B);\ Terminate(j,m)) [\![s_{AB} \mid TSet \mid \{\}]\!] Timers(j,m))) \setminus TSet \end{array} \right) \setminus \{int\}$$

[definition of **par**]

$= (c \rightarrow (\Phi(A)\ \mathbf{par}\ Timers(k,n)) \Box\ Wait\ d;\ int \rightarrow (\Phi(B)\ \mathbf{par}\ Timers(j,m))) \setminus \{int\}$ [hypothesis]

$= (c \rightarrow A \Box\ Wait\ d;\ int \rightarrow B) \setminus \{int\}$ [property of $\Box$ (Law 11) and definition of timeout]

$(c \rightarrow A) \overset{d}{\rhd} B$ $\Box$

Proofs for the remaining cases of the induction can be found in [She06].

4.5. Linking refinement of untimed and timed actions

The main motivation for our normal form is to isolate explicit time concerns in the timers, with the aim of conducting reasoning using only the action resulting from the application of the function Φ. It would be desirable that, for any actions A and B, the compositionality property $(\Phi(A) \sqsupseteq \Phi(B)) \Rightarrow (A \sqsupseteq B)$ held, but this is not true in general: when the timers in the normal forms of A and B are different, they cannot be easily compared. It is reasonable, however, to assume that the timers of a specification and corresponding implementation are the same; if they are not, extra timers can be added. In such cases, the following theorem holds; for simplicity, we omit the parameters of the *Timers* component.

Theorem 4 If, for an action *Timers*, we have that $A = \Phi(A)\ \mathbf{par}\ Timers$ and $B = \Phi(B)\ \mathbf{par}\ Timers$, then $(\Phi(A) \sqsupseteq \Phi(B)) \Rightarrow (A \sqsupseteq B)$.

Proof. Direct from the monotonicity of the **par** operator. $\Box$

This theorem shows that, provided the timers of the normal forms of a specification and an implementation are the same, we can ignore them for reasoning. This is, however, still a result in the *Circus Time* model. To justify reasoning in the untimed *Circus* model, we abstract $\Phi(A)$ and $\Phi(B)$ using the function L. As explained in Sect. 4.3, syntactically, $L(\Phi(A))$ is identical to $\Phi(A)$, since $\Phi(A)$ does not contain any time operators, but $L(\Phi(A))$ is an action in the untimed *Circus* theory, which we want to use for reasoning. The same abstraction can be applied to $\Phi(B)$. The theorem below establishes the soundness of this final step of our validation approach: we can conclude that $\Phi(A)$ refines $\Phi(B)$, by proving that $L(\Phi(A))$ refines $L(\Phi(B))$.

Theorem 5 For any A and B, let $A' = \Phi(A)$ and $B' = \Phi(B)$ then $(L(A') \sqsupseteq L(B')) \Rightarrow (A' \sqsupseteq B')$.

Proof.

$L(A') \sqsupseteq L(B')$ [R is monotonic]

$\Rightarrow R(L(A')) \sqsupseteq R(L(B'))$ [Theorem 1]

$\Rightarrow A' \sqsupseteq R(L(B'))$ [B' is time insensitive]

$= A' \sqsupseteq B'$ $\Box$

In summary, the proposed validation framework can be used as follows. Starting from a timed specification S

and a timed program P, we generate their respective normal forms, with the same set of timers.

$$S = \Phi(S) \textbf{ par } Timers$$

$$P = \Phi(P) \textbf{ par } Timers$$

Afterwards, we abstract $\Phi(S)$ and $\Phi(P)$ into the untimed model and prove that the following holds.

$$L(\Phi(P)) \sqsupseteq L(\Phi(S))$$

From Theorem 5, it follows that $\Phi(P) \sqsupseteq \Phi(S)$. Then, using Theorem 4, we reach the required conclusion.

$$P \sqsupseteq S$$

The fact that the normal form reduction preserves semantics is guaranteed by Theorem 3.

As an example, we verify an implementation *Imp* of our alarm controller as a car alarm system. When enabled, it waits for the elapse of T_1 time units before the alarm can be *disturbed*. It also assures that after detecting a disturbance, it waits for at least T_2 time units before the alarm is triggered.

$$Imp \mathrel{\widehat{=}} \mu\, X \bullet enable \rightarrow Wait\ T_1;\ disturbed \rightarrow Wait\ T_2;\ alarm \rightarrow disable \rightarrow X$$

We can obtain an untimed version of our implementation by applying the Φ function.

$$\Phi(Imp) = \mu\, X \bullet enable \rightarrow setup.1!T_1 \rightarrow out.1.T_1 \rightarrow disturbed \rightarrow setup.1!T_2 \rightarrow out.1.T_2 alarm \rightarrow disable \rightarrow X$$

The timers defined for the above program are the same used for the normalised specification $Alarm_{NF}$ in Sect. 4.1, that is, $Timer_1 \mathrel{\widehat{=}} Timer(1, T_1)$ and $Timer_2 \mathrel{\widehat{=}} Timer(2, T_2)$. Therefore, we prove that the untimed alarm controller $L(\Phi(Imp))$ satisfies its untimed specification $Alarm_{NF}$. Because both actions do not contain time operators, we can use the CSP model checker FDR in the verification. It easily proves that the untimed action $Alarm_{NF}$ is refined by the untimed action $L(\Phi(Imp))$, in the traces model. Consequently, we can then conclude that the timed specification of the alarm is refined by the implementation.

5. Conclusions

In this paper, we present a proposal for extending the UTP framework by adding a theory for discrete time, state-rich reactive systems. The new theory is validated by a detailed comparison with the original CSP and *Circus* theories, and by its use to define a semantics for *Circus Time*. New healthiness conditions are proposed and related to the original UTP healthiness conditions for reactive programs and CSP processes. We also explore here the semantic relation between the *Circus* and the *Circus Time* theories. Finally, we propose a framework for the specification and validation of real-time systems using *Circus Time* as a specification notation; it allows us to use FDR for analysis of timed programs.

Our proposed time theory has inspired others to define languages using the UTP. Ri Hoyn Sul and He Jifeng defined semantics for a timed version of the RAISE Specification Language, namely, Timed RSL, using our theory [SH03]. An interesting contribution of theirs is an interlock composition operator, which is similar to a parallel composition, but differs in the interaction of the processes with the environment.

Our theory has also inspired the development of new models, like that proposed by Qin Shengchao, Jin Song Dong, and Wei-Ngan Chin in [QDC03], which is an extension to our theory to add observations of sensor–actuator variables along with the timed traces; the model was used to give semantics to a subset of TCOZ [MD00]. An interesting aspect of that work is the definition of two additional time operators. The first is used to define a maximum execution time: $P \bullet Deadline\ t$ imposes a time constraint on the process P, which requires it to terminate within time t. The second operator $P \bullet Waituntil\ t$ behaves as follows: if the process P terminates within t time units, then the program waits until t time units have passed before finishing; otherwise, it behaves as P. In [DHQ+04], a set of transformation rules are presented to convert a TCOZ specification into a Timed Automata; then UPPAAL [LPY97] is used to verify time properties.

The work presented in [BSW07] describes slotted *Circus*: a family of UTP theories based on that presented here. It provides different ways of organising the events in each of the elements (time slots) of the timed trace. In our theory, they are organised in sequences, but the family in [BSW07] also allows the use of multisets, for example. This supports the modelling of synchronous languages like Handel-C, in which data and communication operations are in synchrony with a global clock edge marking the end of a computation cycle. The generality

of the family of theories provides further unification and, therefore, paves the way to a reasoning technique that uses the simple untimed model, or different time models, when needed.

For an early effort to encode timed into untimed models, we can refer back to work on converting timed into untimed automata [AD94, ON96]. Based on these results, the approach reported in [CH04] starts with a specification in DC and generates a model in a variation of the DC notation in which formulas correspond to regular expressions over a state of special symbols. These formulas can be easily translated into an untimed automata or, alternatively, into Promela models for analysis using SPIN. This work, however, does not address soundness of the translations, and the change of DC notation does lead to loss of time information.

The idea of timed specification decomposition into untimed specification and timers was introduced by Meyer in [Mey01]. In that work, a strategy to decompose a Timed CSP specification into an untimed CSP specification and a collection of timers is presented. The focus of Meyer's work is on its application within the RT-Tester test system, with the objective of generating test cases, test execution and test evaluation. The timer used by Meyer does not consider timeouts or termination. Here, we have extended the structural decomposition approach proposed by Meyer and applied it to refinement checking.

Ouaknine [Oua01] developed an algorithm for transforming a dense time model of Timed CSP into a discrete time model that is based on a *tock* event and the CSP untimed semantic models. The event *tock* represents the passage of time and has a special semantics captured by a new external choice operator. Standard methods for model checking have been applied with the aid of FDR. The use of the *tock* event was also introduced by Schneider [Sch00], and has a drawback in terms of the size of the models. For example, a command *Wait* 3 is represented as $tock \rightarrow tock \rightarrow tock \rightarrow Skip$. On other hand, in our approach we give a direct semantic representation for the timed program, and then present a normal form where time operators are interpreted by timer events; this means that we only mark the beginning and end of the wait period with timer events without considering the passage of time in between. In this way, we reduce the number of possible states in the verification and make the use of a tool such as FDR much more efficient. Experimental results, however, are not available, and will be the subject of future work.

In a previous work [SSC01], we presented Timed CSP-Z, a language that integrates Timed CSP and Z using the same approach used by CSP-Z [Fis00]. The language was used in the specification of an industrial case study, and rules were provided to convert a Timed CSP-Z specification into a special type of Timed Petri Net known as TBNets [GMMP89], with CABERNET [GP92] used to automate validation [SSC03]. In the approach that we have presented here, we avoid the need for any translation between different formalisms; rather, we have defined Galois connections between the timed and untimed models.

A reasoning framework similar to ours is proposed in [RU05] for μCRL, a process algebra that is based on ACP and includes facilities for equational specification of data. In that work, a timed specification is transformed to a normal form using a technique called linearisation, and then a time-free abstraction is used for reasoning using techniques for untimed μCRL. For example, strong bisimilarity between time-free abstractions establishes bisimilarity for the corresponding timed specifications.

In a future line of work, we will explore other time operators. The timed interrupt operator, in particular, can be defined in terms of *Wait t* and the standard interrupt operator. A UTP semantics for interrupt is presented in [MW09]. An interesting approach to modelling deadlines using miracles is discussed in [Woo09].

Another interesting piece of future work is related to continuous-time models. A major challenge is the healthiness conditions concerning finite variability. These are properties of a (possibly infinite) number of observations of the process: there can only be finitely many events in a finite period of time, and refusal sets record intervals over which sets of events are refused. Just like prefix closure is a challenging healthiness condition for the UTP model of CSP, because it is a property of the set of traces of a process, and not of a pair of observations of its behaviour, we expect that healthiness conditions related to finite variability will be challenging as well. How much we can achieve without considering such healthiness conditions, or how we can overcome the difficulty in expressing them, is an open question at this stage.

In *Circus* much work has been done on the elaboration of refinement laws [CSW03, Oli06]. Our work is a solid foundation to extend the catalogue of *Circus* refinement laws to include both time and untimed programs. It is interesting to observe that most algebraic properties of untimed programs are preserved in the timed model. Before tackling this point, we will apply our validation framework to a number of elaborate case studies with different types of real-time properties.

Acknowledgments

We are grateful to Jim Woodcock for his detailed comments on our work. The work of Augusto Sampaio is partially funded by the Brazilian Research Council (CNPq).

A. More semantic definitions

$$
\begin{array}{rcl}
Stop & \widehat{=} & CSP1_t(R3_t(ok' \wedge wait' \wedge trace' = \langle\rangle)) \\
Chaos & \widehat{=} & R_t(true) \\
x := e & \widehat{=} & CSP1(R_t(ok = ok' \wedge wait = wait' \wedge tr'_t = tr_t \wedge state' = state \oplus \{x \mapsto val(e, state)\})) \\
c!e & \rightarrow & Skip = c.e \rightarrow Skip \\
c?x & \rightarrow & Skip = \exists\, e \bullet c.e \rightarrow Skip[state_o/state] \wedge state' = state_o \oplus \{x \mapsto e\} \\
b\&A & \widehat{=} & A \lhd b \rhd Stop
\end{array}
$$

B. Laws of timed actions

Here we present a small subset of the *Circus Time* laws that are used in the proofs presented in this paper.

Law 20

$$
\begin{array}{l}
(a \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket ((Wait\ d;\ (a \rightarrow A')) \Box (b \rightarrow B)) \\
= \\
Wait\ d;\ (a \rightarrow (A \llbracket s_A \mid cs \mid s_B \rrbracket A'))
\end{array}
$$

provided $a \in cs$ and $b \in cs$.

Law 21 $((a \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket (a \rightarrow A') \Box (b \rightarrow B)) = (a \rightarrow (A \llbracket s_A \mid cs \mid s_B \rrbracket A'))$
provided $a \in cs$ and $b \in cs$.

Law 22 $((a \rightarrow A) \llbracket s_A \mid s_B \rrbracket (b \rightarrow B)) = ((a \rightarrow (A \llbracket s_A \mid s_B \rrbracket (b \rightarrow B))) \Box (b \rightarrow ((a \rightarrow A) \llbracket s_A \mid s_B \rrbracket B)))$

Law 23 $((a \rightarrow A) \llbracket s_A \mid cs \mid s_B \rrbracket B) = a \rightarrow (A \llbracket s_A \mid cs \mid s_B \rrbracket B)$
provided $a \notin cs$ and $initials(B) \subseteq cs$.

Law 24 $((a \rightarrow A) \llbracket s_A \mid cs \mid s_{BC} \rrbracket ((a \rightarrow B) \llbracket s_B \mid s_C \rrbracket C) = (a \rightarrow (A \llbracket s_A \mid cs \mid s_{BC} \rrbracket (B \llbracket s_B \mid s_C \rrbracket C)))$
provided $\{a\} \cup initial(C) \subseteq cs$ and $a \notin initials(C)$.

Law 25 $(c \rightarrow A) \setminus cs = A \setminus cs$
provided $c \in cs$.

Law 26 $(c \rightarrow A) \setminus cs = c \rightarrow (A \setminus cs)$
provided $c \notin cs$.

Law 27 $Wait\ n \setminus cs = Wait\ n$

Law 28 $A \setminus cs = A$
provided $cs \cap usedC(A) = \emptyset$.

Law 29 $(A \setminus cs_1) \setminus cs_2 = A \setminus (cs_1 \cup cs_2)$

Law 30 $(A;\ B) \setminus cs = (A \setminus cs);\ (B \setminus cs)$

Law 31 $((a \rightarrow A) \Box (b \rightarrow B)) \setminus cs = (a \rightarrow (A \setminus cs)) \Box (b \rightarrow (B \setminus cs))$
provided $a \notin cs$ and $b \notin cs$.

Law 32 $(A \llbracket s_A \mid pcs \mid s_B \rrbracket B) \setminus cs = (A \setminus cs \llbracket s_A \mid pcs \mid s_B \rrbracket B \setminus cs)$
provided $pcs \cap cs = \emptyset$.

Law 33 $((a \rightarrow A) \Box (Wait\ d;\ b \rightarrow c \rightarrow B)) \setminus \{b, c\} = ((a \rightarrow A) \Box (Wait\ d;\ c \rightarrow B)) \setminus \{b, c\}$

Law 34 $((a \rightarrow A) \Box (Wait\ d;\ b \rightarrow B)) \setminus cs = ((a \rightarrow (A \setminus cs)) \Box (Wait\ d;\ b \rightarrow (B \setminus cs)))$
provided $a \notin cs$ and $b \notin cs$.

References

[AD94] Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235

[Bey01] Beyer D (2001) Improvements in BDD-based reachability analysis of timed automata. In: FME 2001: formal methods for increasing software productivity. Lecture notes in computer science, vol 2021. Springer, Berlin, pp 318–343

[BH81] Bernstein A, Harter PK (1981) Proving real-time properties of programs with temporal logic. In: ACM symposium on operating system principles, pp 1–11

[BLL+95] Bengtsson J, Larsen KG, Larsson F, Pettersson P, Yi W (1995) UPPAAL—a tool suite for automatic verification of real-time systems. In: Workshop on verification and control of hybrid systems III, number 1066 in Lecture notes in computer science. Springer, Berlin, pp 232–243

[BSW07] Butterfield A, Sherif A, Woodcock JCP (2007) Slotted Circus: A UTP-family of reactive theories. In: International conference on formal enginneering. Lecture Notes in Computer Science. Springer, Berlin

[CH04] Chun KY, Hung DV (2004) Verifying real-time systems using untimed model checking tools. In: Technical report UNU-IIST-TR-3002, The United Nations University, International Institute for Software Engineering

[Che93] Chen L (1993) Timed processes: models, axioms and decidability. PhD thesis, The University of Edinburgh, Department of Computer Science

[Che99] Chellas BF (1999) Modal Logic: An Introduction. Cambridge University Press, London

[CHR91] Chaochen Z, Hoare CAR, Ravn AP (1991) A calculus of duration. Inf Process Lett 40:269–276

[CSW03] Cavalcanti ALC, Sampaio ACA, Woodcock JCP (2003) A refinement strategy for *Circus*. Formal Aspects Comput 15 (2-3):146–181

[CW06] Cavalcanti ALC, Woodcock JCP (2006) A tutorial introduction to CSP in unifying theories of programming. In: Refinement techniques in software engineering, Lecture notes in computer science, vol 3167. Springer, Berlin, pp 220–268

[DHQ+04] Dong JS, Hao P, Qin S, Sun J, Wang Y (2004) Timed patterns: TCOZ to timed automata. In: International conference on formal engineering methods. Lecture notes in computer science. Springer, Berlin

[Dij76] Dijkstra EW (1976) A Discipline of Programming. Prentice-Hall, New Jersey

[DS89] Duke R, Smith G (1989) Temporal logic and z specifications. Aust Comput J 21(2):62–66

[DS95] Davies J, Schneider S (1995) A brief history of Timed CSP. Theor Comput Sci 138(2):243–271

[EHLW97] Evans AS, Holton DRW, Lai LM, Watson P (1997) A comparison of real-time formal specification languages. In: Duke DJ, Evans AS (eds) Northern formal methods workshop, electronic workshops in computer science. Springer, Berlin

[Eva94] Evans AS (1994) Visualising concurrent Z specifications. In: Boen JP, Hall J (eds) Z user workshop, workshops in computing, pp 269–281. Springer, Berlin

[FC06] Freitas AF, Cavalcanti ALC (2006) Automatic translation from *Circus* to Java. In: Misra J, Nipkow T, Sekerinski E (eds) FM 2006: formal methods. Lecture notes in computer science, vol 4085. Springer, Berlin, pp 115–130

[Fis98] Fischer C (1998) How to combine Z with a process algebra. In: Bowen J, Fett A, Hinchey M (eds) ZUM'98: the Z formal specification notation. Springer, Berlin

[Fis00] Fischer C (2000) Combination and implementation of processes and data: from CSP-OZ to Java. PhD thesis, Fachbereich Informatik Universität Oldenburg

[For97] Formal Systems (Europe) Ltd. Failures-divergence refinement, 1997 Revision 2.0.

[Fre06] Freitas LJS (2006) Model Checking *Circus*. PhD thesis, University of York, Department of Computer Science

[GMMP89] Ghezzi C, Mandrioli D, Morasca S, Pezze M (1989) A general way to put time in petri nets. In: Fifth International workshop on Software Specification and Design, pp 60–67. IEEE Computer Society and ACM

[GP92] Ghezzi C, Pezze M (1992) Cabernet: an environment for the specification and verification of real-time systems. In: DECUS Europe Symposium

[HH98] Hoare CAR, He Jifeng (1998) Unifying Theories of Programming. Prentice-Hall

[Hoa85] Hoare CAR (1985) Communicating Sequential Processes. Prentice-Hall, New Jersey

[JM86] Jahanian F, Mok AK (1986) Safety analysis of timing properties in real-time systems. IEEE Trans Softw Eng 12(9):890–904

[HV02] He Jifeng, Verbovskiy V (2002) Integrating CSP and DC. R 248, International Institute for Software Technology, The United Nation University

[Lam94] Lamport L (1994) The temporal logic of actions. ACM Transactions on Programming Languages and Systems 16(3):872–923

[LH99] Li L, Jifeng He (1999) A Denotational Semantics of Timed RSL using Duration Calculus. R 168, International Institute for Software Technology, The United Nation University

[LPY97] Larsen KG, Pettersson P, Yi W (1997) UPPAAL in a Nutshell. International Journal on Software Tools for Technology Transfer

[MD00] Mahony B, Dong JS (2000) Timed Communicating Object Z. IEEE Trans Softw Eng 26(2):150–177

[Mey01] Meyer O (2001) Structural decomposition of timed CSP and its application in real-time testing. Master's thesis, University of Bremen

[Mor94] Morgan CC (1994) Programming from specifications, 2nd edn. Prentice-Hall, New Jersey

[Mur89] Murata T (1989) Petri nets: properties, analysis and applications. Proc IEEE 77(4):541–580

[MW09] McEwan A, Woodcock JCP (2009) Unifying Theories of Interrupts. In: Unifying Theories of Programming 2008. Lecture notes in computer science. Springer, Berlin

[OCW07a] Oliveira MVM, Cavalcanti ALC, Woodcock JCP (2007) A UTP Semantics for *Circus*. Formal Aspects of Computing, online first

[OCW07b] Oliveira MVM, Cavalcanti ALC, Woodcock JCP (2007) Unifying Theories in ProofPowerZ. Formal Aspects of Computing, online first

[Oli06] Oliveira MVM formal derivation of state-rich reactive programs using *Circus*. PhD thesis, University of York, 2006

[ON96] Ostro J, Ng H (1996) Verifying real-time systems with standard tools. In: AMAST workshop on real-time systems

[Oua01] Ouaknine J (2001) Discrete analysis of continuous behaviour in real-time concurrent systems. PhD thesis, Oxford University

[Pnu77] Pnueli A (1977) The temporal logic of programs. In: 18th IEEE symposium foundations of computer science, pp 46–57
[QDC03] Qin S, Dong JS, Chin WN (2003) A semantic foundation for TCOZ in unifying theories of programming. In: Araki K, Gnesi S, Mandrioli D (eds) FME2003: formal methods. Lecture notes in computer sscience, vol 2805. Springer, Berlin, pp 321–340
[Ros98] Roscoe AW (1998) The theory and practice of concurrency. Prentice-Hall series in computer science. Prentice-Hall, New Jersey
[RR88] Reed GM, Roscoe AW (1988) A timed model for communicating sequential processes. Theor Comput Sci 58:249–261
[RU05] Reniers MA, Usenko YS (2005) Analysis of timed processes with data using algebraic transformations. In: International symposium on temporal representation and reasoning, pp 192–194. IEEE Computer Society
[Sch00] Schneider S (2000) Concurrent and Real-time Systems: The CSP Approach. Wiley, London
[She06] Sherif A (2006) A framework for specification and validation of real-time systems using *Circus* actions. PhD thesis, Centro de Informática/UFPE, Brazil
[SH02] Sherif A, He Jifeng (2002) Towards a time model for *circus*. In: George C, Miao H (eds) International conference on formal engineering methods. Lecture notes in computer sscience, vol 2495. Springer, Berlin, pp 613–624
[SH03] Sul RH, He Jifeng (2003) A complete verification system for timed RSL. R 275, International Institute for Software Technology, The United Nation University
[SHCS05] Sherif A, He Jifeng, Cavalcanti ALC, Sampaio ACA (2005) A framework for specification and validation of real-time systems using *circus* actions. In: Liu Z, Araki K (eds) International colloquium on theoretical aspects of computing.Lecture notes in computer science, vol 3407. Springer, Berlin, pp 478–493
[SSC01] Sherif A, Sampaio ACA, Cavalcante S (2001) An integrated approach to specification and validation of real-time systems. In: Formal Methods Europe. Lecture notes in computer science, vol 2021. Springer, Berlin, pp 278–299
[SSC03] Sherif A, Sampaio ACA, Cavalcante S (2003) Specification and validation of the saci-1 on-board computer using timed-csp-z and petri nets. In: International conference On application and theory of Petri Nets, pp 161–180
[TM87] Turski WM, Maibaum TSE (1987) The specification of computer systems. International computer science series. Addison-Wesley, Reading
[Toy02] Toyn I, (ed) (2002) Information technology—Z formal specification notation—syntax, type system and semantics. ISO, ISO/IEC 13568:2002(E)
[TS99] Treharne H, Schneider S (1999) Using a process algebra to control B OPERATIONS. In: 1st International conference on integrated formal methods—IFM 1999, pp 437–457. Springer, Berlin
[WC02] Woodcock JCP, Cavalcanti ALC (2002) The semantics of *Circus*. In: Bert D, Bowen JP, Henson MC, Robinson K (eds) ZB 2002: formal specification and development in Z and B. Lecture Notes in Computer Science, vol 2272. Springer, Berlin, pp 184–203
[WCF05] Woodcock JCP, Cavalcanti ALC, Freitas L (2005) Operational semantics for model-checking *Circus*. In: Fitzgerald J, Hayes IJ, Tarlecki A (eds) FM 2005: formal methods. Lecture notes in computer science, vol 3582. Springer, Berlin, pp 237–252
[WD96] Woodcock JCP, Davies J (1996) Using Z—specification, refinement, and proof. Prentice-Hall, New Jersey
[Woo09] Woodcock JCP (2009) The miracle of reactive programming. In: Butterfield A (ed) Unifying theories of programming 2008, Lecture notes in computer science. Springer, Berlin

Received 6 January 2009
Accepted in revised form 14 June 2009 by Dong Jin Song and C. B. Jones
Published online 15 July 2009