Constructing Property-Oriented Models for Verification

He Jifeng[1⋆], Shengchao Qin[2⋆⋆], and Adnan Sherif[3]

[1] Software Engineering Institute, East China Normal University
jifeng@sei.ecnu.edu.cn
[2] Department of Computer Science, Durham University
shengchao.qin@durham.ac.uk
[3] Centro de Informatica, Federal University of Pernambuco

Abstract. This paper advocates a general approach to formal verification by constructing property-oriented models. We instantiate the approach using timing properties, and construct a heterogeneous untimed model in which time is abstracted away, so that we can verify timing properties in an untimed framework. The correctness of property-oriented model construction is ensured by the conformance of semantic and syntactic mappings.

1 Introduction

It has been noticed that a single software development method is not sufficient to solve all types of problems found in complex software systems. The integration of software development methods has been proposed and investigated in the recent years, for example, the integration of state-modeling and process languages has become an active area of research ([19, 6, 1, 11]). Such blending of different notations can provide us more powerful languages for specifying very complex software systems. Unified observation-oriented models behind the integrated languages (like [14], [23]) can ensure the soundness of the integration of different notations, and can be used as a reference document for developing tool supports. However, such complete models are usually very complicated and thus hard to use for the verification purpose.

Properties to be verified or analysed can be divided into different categories, each kind of properties only refer to part of the whole observation model, such as safety properties that are not time dependent, timing properties, deadlock-free properties. Recent work [4] suggests a projection approach to the verification of timing properties. The projection can be conducted in a syntax-directed manner, where the soundness proof replies on a deep projection from the whole model to the sub-model, thus the whole model should be built first, which is usually very time-consuming. Therefore, we propose to construct (small) property-oriented models for the verification of any particular kind of properties. We shall guarantee that different property-oriented sub-models can be integrated into the whole model in a later stage, where necessary. In this paper, we elaborate this general idea using timing properties. We construct an untimed heterogeneous model, where time information is abstracted away, and handled by a special

⋆ Supported in part by China 973 Project 2002CB312001.
⋆⋆ Author for Correspondence. Supported in part by China NNSFC Project 60573081.

Timer process. With such a property-oriented model we can verify certain kind of timing properties using the simpler untimed model, either by model checking or theorem proving. This greatly simplifies the verification process.

We demonstrate our approach using a small language Cz, which is a subset of the combination of CSP [7] and Z [20]. It can be regarded as a subset of the *Circus* [21] language, or a subset of another powerful specification language TCOZ [11]. We shall focus on timing properties that can be described in programming languages, rather than specification languages, like:

- the delay between two consecutive events should at least be t units of time;
- a program awaits an event at most t units of time before it does something else.

More general timing properties that can be described in specification languages but difficult in programming languages, like deadline and waituntil in TCOZ, will not be covered here.

This paper makes the following contributions:

- We propose a general approach to verification by constructing *property-oriented* models for integrated formal languages.
- We demonstrate our approach in terms of timing properties. We build an untimed model for the verification of real-time properties.
- We build a deep link between timed traces and untimed heterogeneous traces (with timer events). From that, we can generate the provably-correct untimed model.
- We illustrate our approach through an alarm controller example.

The rest of the paper is organized as follows. Section 2 introduces the illustrative example. Section 3 describes the language model. The approach is presented in detail in Section 4, followed by related work and conclusion.

2 An Illustrative Example

In this section, we use a small example to illustrate a novel approach to the verification of timing properties for reactive systems.

2.1 The Alarm Controller

The alarm system was first used in [9]. The system is a common alarm controller that can be found in buildings and cars. The controller is connected to a sensor which detects movements or changes in the environment monitored by the alarm. The controller operates in two modes: when disabled, it will ignore any disturbance detected by the sensor; when enabled, the controller will sound an alarm when the sensor signals a disturbance.

There are two timing requirements on the alarm controller: the first states that after the controller is enabled, there is a period of t_1 units of time before a disturbance can cause the alarm to ring. The period t_1 permits a person to enable the alarm and then leave without causing it to sound. The second requirement states that when a detected disturbance is received, the controller will wait for another period of t_2 units of time

before activating the alarm. The period t_2 leaves some time to the legal user to disable it before it sounds.

Let us analyse the first timing requirement, that is, when the controller is enabled, there is a delay of t_1 units of time before it can receive any disturbance from its sensor. As a first attempt, we can specify this requirement in terms of the following action:

$$R_1 \mathrel{\widehat{=}} \mathit{enable} \rightarrow \mathit{Wait}\ t_1; \mathit{disturb} \rightarrow R$$

Notice the event *enable* indicates the alarm system is enabled, while the event *disturb* denotes a disturbance detected by the sensor. At this moment, we ignore the subsequent behaviour after a disturbance is received and simply use R to denote it.

The key idea of our approach is to separate timing properties from logical properties by introducing a specific component, called *Timer*, to take care of the timing features. Thus we can use existing untimed verification tools like model checkers to verify that certain time properties are met, rather than construct a new tool for verification from scratch.

For R_1, we can transform it to the following untimed action:

$$R_1' \mathrel{\widehat{=}} \mathit{enable} \rightarrow \mathit{set}! \rightarrow \mathit{reset}? \rightarrow \mathit{disturb} \rightarrow R$$

The two new events *set* and *reset* are used to interact with the following *Timer* action:

$$\mathit{Timer} \mathrel{\widehat{=}} \mathit{set}? \rightarrow \mathit{Wait}\ t_1; \mathit{reset}! \rightarrow \mathit{Skip}$$

Note that the *Timer* component is in charge of time control. It is activated by *set* signal, and after t_1 time elapses, it notifies the process R_1' via signal *reset*.

To verify R_1 meets the property that a disturbance can only be received after the controller is enabled for t_1 units of time (we refer to it as t_1-*delay property* in what follows), we only need to check the following untimed property for R_1':

$$\begin{aligned}\forall \mathit{utr}_0, \mathit{utr}_1, \mathit{utr}_2 \cdot ((\mathit{utr} = \mathit{utr}_0 \frown \langle \mathit{enable} \rangle \frown \mathit{utr}_1 \frown \langle \mathit{disturb} \rangle \frown \mathit{utr}_2 \ \wedge \\ \mathit{utr}_1 \upharpoonright \{\mathit{enable}, \mathit{disturb}\} = \langle\rangle) \ \Rightarrow \ \mathit{utr}_1 = \langle \mathit{set}, \mathit{reset} \rangle)\end{aligned}$$

It states that there are only two timer events *set* and *reset* between an *enable* event and its consecutive *disturb* event. The event *set* activates the timer, while *reset* deactivates the timer, which indicates t_1 time is passed. Together with the timer action, it ensures the t_1-delay property. Note that *utr* denotes the (untimed) trace, i.e. a sequence of events, while utr_i's are segments of the trace. Formal definitions will be given in a later section.

The soundness for the separation of timing features from logical features can be specified in terms of the following equation:

$$R_1 = (R_1' [\![\{\mathit{set}, \mathit{reset}\}]\!] \mathit{Timer}) \setminus \{\mathit{set}, \mathit{reset}\}$$

This can be easily proved using the expansion laws for parallel composition. The right hand side is a parallel composition of an untimed action (R_1') and a timer action (*Timer*) which communicate with each other via two internal events *set* and *reset* (hidden from outside). Such a parallel composition is the normal form we shall adopt for verification.

2.2 The Normal Form

In this subsection, we shall deal with the complete specification for the alarm controller.

The complete timed specification for the alarm controller is given as follows.

$$
\begin{aligned}
\mathit{Disable} &\mathrel{\widehat{=}} \mathit{disable} \rightarrow \mathit{Skip} \\
\mathit{Running} &\mathrel{\widehat{=}} \mathit{Disable} \,\Box\, (\mathit{disturb} \rightarrow \mathit{Active}) \\
\mathit{Active} &\mathrel{\widehat{=}} \mathit{Disable} \overset{t_2}{\rhd} (\mathit{alarm} \rightarrow \mathit{Disable}) \\
\mathit{Alarm} &\mathrel{\widehat{=}} \mu X \bullet \mathit{enable} \rightarrow (\mathit{Disable} \overset{t_1}{\rhd} \mathit{Running}); X
\end{aligned}
$$

Note that event *disable* is used to disable the controller, event *alarm* signals the firing of the alarm. For more flexibility, we allow the controller to be disabled at any point during running. We use timeout constructs (defined later in Section 3.2) to capture this requirement. Take *Alarm* as an example, once the controller is enabled, it is either disabled (and then waits for *enable* again), or is ready to receive any disturbance after t_1 (*Running*).

As explained in last subsection, we shall transform the timed specification *Alarm* to a normal form composed of an untimed specification in parallel with a *Timer* action.

We shall use function Φ to abstract away timing features from a timed action. The complete definition for Φ will be given when we present the syntax of the language. The following is the result after applying it to the above specification.

$$
\begin{aligned}
\Phi(\mathit{Disable}) &\mathrel{\widehat{=}} \mathit{disable} \rightarrow \mathit{Skip} \\
\Phi(\mathit{Running}) &\mathrel{\widehat{=}} \Phi(\mathit{Disable}) \,\Box\, (\mathit{disturb} \rightarrow \Phi(\mathit{Active})) \\
\Phi(\mathit{Active}) &\mathrel{\widehat{=}} \mathit{set}!t_2 \rightarrow \\
&\quad ((\mathit{disable} \rightarrow \mathit{halt}! \rightarrow \mathit{Skip}) \,\Box\, (\mathit{reset}? \rightarrow \mathit{alarm} \rightarrow \Phi(\mathit{Disable}))) \\
\Phi(\mathit{Alarm}) &\mathrel{\widehat{=}} \mu X \bullet \mathit{enable} \rightarrow \mathit{set}!t_1 \rightarrow \\
&\quad ((\mathit{disable} \rightarrow \mathit{halt}! \rightarrow \mathit{Skip}) \,\Box\, (\mathit{reset}? \rightarrow \Phi(\mathit{Running})); X
\end{aligned}
$$

Note that a new timer event *halt* is used to stop the timer when event *disable* arrives during the ticking of the clock. There are two timing requirements in the specification, thus we design the general timer action as follows:

$$\mathit{Timer} \mathrel{\widehat{=}} \mu X \bullet \mathit{set}?x \rightarrow ((\mathit{halt}? \rightarrow \mathit{Skip}) \,\Box\, (\mathit{Wait}\ x; \mathit{reset}! \rightarrow \mathit{Skip})); X$$

Note that when the timer is *set* to work, a value is passed to it (stored in x) to indicate the time duration that it should count before it generates a *reset* signal.

Now the timed specification *Alarm* is transformed to the following normal form:

$$(\Phi(\mathit{Alarm}) || [\{\mathit{set}, \mathit{reset}, \mathit{halt}\}] || \mathit{Timer}) \setminus (\{\mathit{set}, \mathit{reset}, \mathit{halt}\}$$

The following theorem ensures the soundness of the abstraction.

Theorem 1. *We have*

$$\mathit{Alarm} \;=\; (\Phi(\mathit{Alarm}) || [\{\mathit{set}, \mathit{reset}, \mathit{halt}\}] || \mathit{Timer}) \setminus \{\mathit{set}, \mathit{reset}, \mathit{halt}\}$$

It is proved using parallel expansion and hiding laws.

2.3 Verification of Timing Properties in the Untimed Model

In this subsection we shall demonstrate that timing requirements can be verified in the untimed framework.

There are two timing requirements for the alarm controller, namely,

- once enabled, the controller should wait at least t_1 units of time before it can receive any disturbance from the sensor.
- once a disturbance is received, the controller should wait at least t_2 units of time before it fires the alarm.

As timing is controlled by the timer actions in our normal form, we can abstract away timing from the above requirements by adding in timer events that are in charge of activating/deactivating the timers. The timing requirements are thus specified in terms of timer events as follows:

$$R_1(utr) \mathrel{\widehat{=}} \forall utr_0, utr_1, utr_2 \cdot (utr = utr_0 \frown \langle enable \rangle \frown utr_1 \frown \langle disturb \rangle \frown utr_2 \ \wedge \\ utr_1 \upharpoonright \{enable, disturb\} = \langle\rangle) \Rightarrow (utr_1 = \langle set.t_1, reset \rangle)$$

$$R_2(utr) \mathrel{\widehat{=}} \forall utr_0, utr_1, utr_2 \cdot (utr = utr_0 \frown \langle disturb \rangle \frown utr_1 \frown \langle alarm \rangle \frown utr_2 \ \wedge \\ utr_1 \upharpoonright \{disturb, alarm\} = \langle\rangle) \Rightarrow (utr_1 = \langle set.t_2, reset \rangle)$$

The overall timing requirement for the alarm controller is thus as below:

$$Req(utr) \mathrel{\widehat{=}} R_1(utr) \wedge R_2(utr)$$

To verify the timed specification (*Alarm*) meets the timing requirements, we only need to demonstrate that the untimed specification ($\Phi(Alarm)$) meets the above requirement, that is, $\Phi(Alarm) \Rightarrow Req(utr)$.

Theorem 2. *Suppose $\Phi(Alarm)$ and $Req(utr)$ are given as above, we have*

$$[\![\Phi(Alarm)]\!] \Rightarrow Req(utr)$$

where $[\![P]\!]$ denotes the observation-oriented semantics for program P.

Proof From the definition of $\Phi(Alarm)$,

$$\Phi(Alarm) = P; \Phi(Alarm)$$

where $P \mathrel{\widehat{=}} enable \rightarrow set!t_1 \rightarrow ((disable \rightarrow halt! \rightarrow Skip) \,\Box\, (reset? \rightarrow \Phi(Running)))$.

Thus the semantic predicate $[\![\Phi(Alarm)]\!]$ is subject to

$$[\![\Phi(Alarm)]\!] = [\![P]\!]; [\![\Phi(Alarm)]\!]$$

That is, it is the fixed point of the equation $X = \mu X \cdot ([\![P]\!]; X)$.

Note that we also use the operator (;) to represent the concatenation of two observational predicates. The formal definition is given in [8].

Due to the following fixed point theorem ([8]):

$$F(S) \sqsupseteq S \ \text{ implies } \ \nu X \cdot F(S) \sqsupseteq S$$

and the fact that there is only one fixed point in this case, we only need to prove

$$[\![P]\!]; Req(utr) \Rightarrow Req(utr)$$

It is thus straightforward as all possible traces of P lie in the following set:

$$\begin{aligned} \mathit{trace}(P) = \{ & \langle \mathit{enable}, \mathit{set}.t_1, \mathit{disable} \rangle,\ \langle \mathit{enable}, \mathit{set}.t_1, \mathit{disable}, \mathit{halt} \rangle, \\ & \langle \mathit{enable}, \mathit{set}.t_1, \mathit{reset}, \mathit{disable} \rangle,\ \langle \mathit{enable}, \mathit{set}.t_1, \mathit{reset}, \mathit{disturb}, \mathit{set}.t_2, \mathit{disable} \rangle, \\ & \langle \mathit{enable}, \mathit{set}.t_1, \mathit{reset}, \mathit{disturb}, \mathit{set}.t_2, \mathit{disable}, \mathit{halt} \rangle \} \\ & \cup \{ \mathit{utr} \mid \mathit{utr} \preceq \langle \mathit{enable}, \mathit{set}.t_1, \mathit{reset}, \mathit{disturb}, \mathit{set}.t_2, \mathit{reset}, \mathit{alarm}, \mathit{disable} \rangle \} \end{aligned}$$

□

Note that the proof is much simpler and more straightforward in comparison with the existing proof given in [9] due to the property-oriented model we use.

3 The Language

This section introduces our language Cz that we use to instantiate our method. We shall give both the untimed and timed models.

3.1 The Untimed Model

The syntax for Cz is given in Fig. 1.

$$\begin{array}{lll} \mathit{Action} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \\ & & \mid \mathit{Communication} \rightarrow \mathit{Action} \mid \mathsf{b}\&\mathit{Action} \\ & & \mid \mathit{Action}; \mathit{Action} \mid \mathit{Action} \Box \mathit{Action} \mid \mathit{Action} \sqcap \mathit{Action} \\ & & \mid \mathit{Action} |[E]| \mathit{Action} \mid \mathit{Action} \backslash E \mid \mu X \bullet \mathit{Action} \\ & & \mid \mathit{Command} \\ \mathit{Command} & ::= & x := e \mid \mathit{Action} \lhd b \rhd \mathit{Action} \\ \mathit{Communication} & ::= & c?[x] \mid c![e] \mid c[.e] \end{array}$$

Fig. 1. Cz : the untimed model

Note that e represents an expression, while b a boolean expression. The set E denotes channel names. The notation $[u]$ indicates that term u is optional.

Skip is a basic action that terminates immediately. *Stop* represents an abnormal termination which simply puts a program in an ever waiting state. *Chaos* is the worst action, nothing can be said about its behaviour. In a *guarded action* ($b\&\mathit{Action}$), the action is preceded by a predicate which has to be true for the action to take place, otherwise the guarded action behaves as *Stop*. An *internal choice* between two actions ($\mathit{Action} \sqcap \mathit{Action}$) selects one of the two actions in a non-deterministic manner, whereas the *external choice* ($\mathit{Action} \Box \mathit{Action}$) waits for any of the two actions to interact with the environment. The first action that interacts with the environment (either by synchronising on an event or terminating) is the resulting action.

The *sequential composition* of two actions ($\mathit{Action}; \mathit{Action}$) behaves as the first action, followed immediately by the second action upon termination of the first. An action can be prefixed with a communication event (input or output) which will take place before the action starts. The action waits for the other actions that need to synchronise on

the channel before the communication can take place. The *parallel composition* of two actions (*Action*$|[cs]|$*Action*) involves a set (*cs*) containing the events they need to synchronise on. A *hiding* operation also takes a set of events (*cs*). The set is to be excluded from the resulting observation of the action, hidden events can no longer be seen by other actions.

An observation-oriented model for the *Circus* language based on Hoare and He's Unifying Theories of Programming [8] is explored in detail in [23, 21], while the unified model for TCOZ is reported in [14]. As our language Cz is a subset of the above two languages, we can borrow the following observation variables from them.

- $ok, ok' : \mathsf{Boolean}$. When *ok* is *true*, it states that the program has started and $ok' =$ *true* indicates that the program has terminated or is in an intermediate stable state.
- $wait, wait' : \mathsf{Boolean}$. When *wait* is *true*, the program starts in an intermediate state. When $wait'$ is *true* the program has not terminated; when it is *false*, it indicates a final observation.
- $state, state'$ are mappings from program variable names to values. The undashed variable represents the initial valuation of the program variables, while the dashed one denotes the valuation at the final observation.
- $utr, utr' : \mathsf{seq}\, Event$ are the sequence of observations on the program's interactions with its environment. *utr* denotes the observations that occur before the program starts, and utr' the final observation. Each element of the sequence is an event.
- $ref, ref' : \mathbb{P}\, Event$ stands for the set of events the program can refuse.

A single observation is given by the combination of the above variables. A program is given as predicates over the observation variables. We give the semantics for basic actions and communication events in what follows to show the use of the above semantic variables. Readers can refer to [23, 14] for the complete set of semantic definitions.

Basic Action The semantics of the action *Skip* is given as a program that can only terminate normally and has no interaction with the environment.

$$[\![Skip]\!] \mathrel{\widehat=} ok' \land \neg wait' \land utr' = utr \land state' = state$$

The action *Stop* is given as a predicate that waits for ever; it does not change the state.

$$[\![Stop]\!] \mathrel{\widehat=} ok' \land wait' \land utr' = utr$$

The assignment attributes a value to a variable in the current state. If the variable does not exist it will be added, otherwise its value will be over written. The assignment operation is instantaneous and does not consume time.

$$[\![x := e]\!] \mathrel{\widehat=} ok' \land \neg wait' \land utr' = utr \land state' = state \oplus \{x \mapsto e\}$$

Note that we abuse the same e in the right hand side to denote the value of e in *state*.

Communication An action can engage in a communication if all the other actions involved in the same communication are ready to do so. We model this with the help of two predicates. $wait\_com(c)$ models the waiting state of an action to communicate on channel c. The only possible observation is that the communication channel cannot

appear in the refusal set during the observation period. *term_com*$(c.e)$ represents the act of communicating a value e over a channel c.

$$\begin{aligned} \textit{wait\_com}(c) &\mathrel{\widehat=} \textit{ok}' \wedge \textit{wait}' \wedge c \notin \textit{ref}' \wedge \textit{utr}' = \textit{utr} \\ \textit{term\_com}(c.e) &\mathrel{\widehat=} \textit{ok}' \wedge \neg \textit{wait}' \wedge \textit{utr}' = \textit{utr} \frown \langle c \rangle \end{aligned}$$

The semantics of the output command is given below.

$$[\![c!e]\!] \mathrel{\widehat=} \textit{wait\_com}(c) \ \vee \ \textit{term\_com}(c.e) \wedge \textit{state}' = \textit{state}$$

The input command can be defined in a similar manner.

$$[\![c?x]\!] \mathrel{\widehat=} \textit{wait\_com}(c) \ \vee \ \textit{term\_com}(c.e) \wedge \textit{state}' = \textit{state} \oplus \{x \mapsto e\}$$

The semantics of the communication prefix can be given in terms of communication and of the sequential composition. The action *comm* is either an input or an output event, or an abstract event name.

$$[\![\textit{comm} \rightarrow \textit{Action}]\!] \mathrel{\widehat=} [\![\textit{comm}; \textit{Action}]\!]$$

3.2 The Timed Model

The timed language TCZ extends the untimed language CZ with two new time operators given in Fig. 2.

$$\begin{aligned} \textit{Action} ::= {} & \cdots \\ & \mid \textit{Wait}\ t \ (\text{time delay}) \\ & \mid \textit{Action} \overset{t}{\rhd} \textit{Action} \ (\text{timeout}) \end{aligned}$$

Fig. 2. TCZ: the timed model

The action $(\textit{Wait}\ t)$ will delay the system for an amount of time determined by the positive integer expression t before terminating normally. The timeout construct $(\textit{Action} \overset{t}{\rhd} \textit{Action})$ takes a positive integer value as the length of the timeout. The timeout operator acts as a time guarded choice. It behaves as either the first or the second action. If the first action performs an observable event or terminates before the specified time elapses, it is chosen. Otherwise, the first action will be suspended and the only possible observations are those produced by the second action.

The semantics for the timed language is given with the same observation variables $\textit{ok}, \textit{ok}', \textit{wait}, \textit{wait}', \textit{state}$ and $\textit{state}'$, while the variables $\textit{utr}, \textit{utr}'$ and $\textit{ref}'$ are replaced by a new pair of variables $\textit{ttr}, \textit{ttr}'$ denoting communication traces in the timed model.

The variable *ttr* records the observations of communication events that occur before the program starts, and $\textit{ttr}'$ records the final observation. Each element of the sequence represents an observation in one time unit. Each observation element is composed of a tuple, where the first element of the tuple is the sequence of events that occur in the

time unit, and the second is the associated set of refusals at the end of the same time unit.

$$ttr, ttr' : \mathsf{seq}(\mathsf{seq}\, Event \times \mathbb{P}\, Event)$$

We maintain an auxiliary variable *utr* that represents a sequence of events that have occurred since the last observation. In this observation we are interested in recording only the events without time.

$$utr : \mathsf{seq}\, Event$$
$$utr = flat(ttr') - flat(ttr)$$

where $flat : \mathsf{seq}(\mathsf{seq}\, Event \times \mathbb{P}\, Event) \rightarrow \mathsf{seq}\, Event$
$flat(\langle\rangle) = \langle\rangle$
$flat(\langle el, ref\rangle \frown S) = el \frown flat(S)$

We show the use of these new variables in the definition of the (*Wait* d) action. The only possible behaviour for this action is to wait for the specified number of units of time to pass before terminating immediately.

$$[\![Wait\ d]\!]_{time} \mathrel{\widehat{=}} ((ok' \wedge wait' \wedge (\#ttr' - \#ttr) < d) \\ \vee (ok' \wedge \neg wait' \wedge (\#ttr' - \#ttr) = d)) \wedge utr = \langle\rangle$$

The timeout action can be defined in terms of external choice as in [17]. The following is a direct definition.

$$[\![P \overset{t}{\rhd} Q]\!]_{time} \mathrel{\widehat{=}} (P \wedge utr = \langle\rangle \wedge \#ttr' - \#ttr \leq t) \vee \\ (\exists k : \#ttr < k \leq \#ttr + t, \exists \tilde{ttr} \bullet \pi_1(ttr'(k)) \neq \langle\rangle \wedge ttr \preceq \tilde{ttr} \wedge \#\tilde{ttr} - \#ttr = k \wedge \\ (\forall i : \#ttr < i < \#ttr + k \bullet \pi_1(ttr'(i)) = \langle\rangle \wedge \tilde{ttr}(i) = ttr'(i)) \wedge P[\tilde{ttr}/ttr]) \vee \\ (\exists \tilde{ttr} \bullet ttr \preceq \tilde{ttr} \wedge \#\tilde{ttr} - \#ttr = t \wedge \\ (\forall i : \#ttr < i < \#ttr + t \bullet \pi_1(ttr'(i)) = \langle\rangle \wedge \tilde{ttr}(i) = ttr'(i)) \wedge Q[\tilde{ttr}/ttr])$$

Note that if P is ready to react to the environment exactly when it has waited for time t, the timeout process chooses P or Q non-deterministically.

Given the semantic model for a TCZ program, we can use the linking function given in [17] to abstract away time information, and thus obtain the corresponding untimed model. This abstraction is useful when we are interested in the verification of time-independent safety properties. In this paper, we shall not elaborate on this aspect but focus more on timing properties.

4 The Approach

This section is devoted to the general approach that we propose to the verification of real-time systems. The verification framework is given in Fig. 3.

Fig. 3 shows us two different approaches. The first one is a top down approach where we start with a timed program and we are interested in checking if the timed program satisfies the time requirements. The second approach is a bottom up method

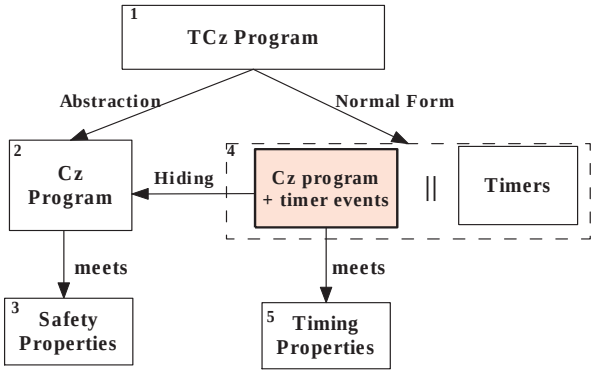

Fig. 3. The Property-Oriented Models for Verification

where we start with an untimed program and add time information where requested. The need for the second one is due to the fact that system development is usually done in stages, in the early stages of development the system designer concentrates on the behaviourial/logical properties of the system, while leaving timing requirement to a later stage. Another aspect is that in some cases it is necessary to identify the hardware used in the implementation to have a clear understanding of the time delays and timeouts that can occur in the system and the points in which they may occur. In the rest of this section we present our approach in detail.

4.1 The Validation Approach

This approach is concerned with the validation of the time requirements of the system using the untimed model. Fig. 3 illustrates the steps for using the framework. The steps to carry out the validation of the time requirements are summarised as follows.

- We start with a specification of our system in the timed model using the timed version of the language. The system designer gives a complete description of the system. All the operators of the language can be used at this stage including parallel composition The timed semantic model of the language is used in this step.
- If we need to verify untimed safety properties, we can use an abstraction function (e.g. the one given in [17]) to obtain an untimed version of the original specification. Such an untimed program can be used to validate the behaviour requirements and safety requirements that are not time dependent. We do not elaborate this aspect in this paper.
- With the help of the normal form function Φ, we obtain a version of the program that has the same semantics as the original program but contains internal timer events. In this step the expansion laws should be used as well to remove all the parallel compositions.
- Time requirements can be expressed in the untimed model with timer events. We can use this untimed model to prove the design meets the time requirements. This can be done using theorem provers or existing (untimed) model checkers.

4.2 The Normal Form

Usually timed programs are implemented with timers, this can be either the system clock or a dedicated timer. Following the same criteria we give a normal form for the time operators. The implementation of the time operators is given as a timer and an untimed program that is synchronised with the timer on dedicated events.

As mentioned above, the events *set*, *halt*, and *reset* are timer events, used by the program to synchronise with and control a timer. The following is the timer program:

$$\mathit{Timer} \mathrel{\widehat=} \mu X \bullet \mathit{set}?x \rightarrow ((\mathit{halt}? \rightarrow \mathit{Skip}) \Box (\mathit{Wait}\ x; (\mathit{reset}! \rightarrow \mathit{Skip})); X$$

The timer is initiated with the event *set* which serves as a trigger. The behaviour of the timer is as follows: after set by its environment, it waits for the *set* signal to set the timer again if an interrupt event *halt* arrives before the timeout, or it emits a signal *reset* and starts to wait for another *set* after it has counted for the designated period of time (stored in x) set by the environment. The event *reset* is similar to other events used in the language, whereas the events *set* and *halt* have special properties. We shall explore these differences further in the next subsection.

Given the definition for a timer action, we aim to generate a function Φ that takes as input a timed program and returns the corresponding untimed program with timer events. That is, for any sequential program *P*, the function Φ should satisfy the following equation:

$$(\dagger) \qquad P \ = \ \Phi(P)\ \mathbf{par}\ \mathit{Timer}$$

Given actions *X* and *Y*, such that $\alpha X \cap \alpha Y = \{\mathit{set}, \mathit{reset}, \mathit{halt}\}$,

$$X\ \mathbf{par}\ Y \mathrel{\widehat=} (X\ |[\{\mathit{set}, \mathit{reset}, \mathit{halt}\}]|\ Y) \backslash \{\mathit{set}, \mathit{reset}, \mathit{halt}\}$$

Note that in the action $(X\ \mathbf{par}\ Y)$, *X* behaves as a master action, while *Y* acts as a slave action. The overall action terminates if and only if the master action *X* terminates.

As timers are not allowed to be shared by parallel actions, several timer actions are needed in the equation $(\dagger)$ in case that *P* is a parallel action.

We shall first give a mapping ψ to abstract away time information from timed traces, while adding timer events properly. This can be regarded as a deep semantic link between a timed process and a heterogeneous communicating process. We only need to define the mapping ψ on *maximal traces*.

Definition 1. *A timed trace* ttr_0 *from a prefix-closed trace set is* maximal, *if for any trace* ttr_1 *that satisfies* $\#\mathit{ttr}_0 = \#\mathit{ttr}_1$, *and* $\forall i : 1..\#\mathit{ttr}_0 \cdot \pi_1(\mathit{ttr}_0(i)) = \pi_1(\mathit{ttr}_1(i))$, *we have* $\forall i : 1..\#\mathit{ttr}_0 \cdot \pi_2(\mathit{ttr}_0(i)) \supseteq \pi_2(\mathit{ttr}_1(i))$.

Definition 2. *Given a set of timed traces TTR, and a single prefix* s, *we define a set of timed traces "after"* s *as follows:*

$$\mathit{TTR}/s \mathrel{\widehat=} \{\mathit{ttr} \mid (s \frown \mathit{ttr}) \in \mathit{TTR}\}$$

Given a trace ttr, we use $\mathit{pref}(\mathit{ttr})$ *to denote the prefix-closed set of traces made out of all prefixes of ttr. We extend it to a set of traces TTR as*

$$\mathit{pref}(\mathit{TTR}) \mathrel{\widehat=} \bigcup \{\mathit{pref}(\mathit{ttr}) \mid \mathit{ttr} \in \mathit{TTR}\}$$

Given two set of traces TTR_1, and TTR_2, the concatenation of them is given as

$TTR_1 \frown TTR_2 \mathrel{\widehat{=}} \{ttr_1 \frown ttr_2 \mid ttr_i \in TTR_i,\ for\ i = 1, 2\}$

Definition 3 (Semantic Mapping). *Let $\mathcal{A}$ denote the maximal set of events of interest, and $\mathcal{A}^*$ denote the sequence closure over $\mathcal{A}$. Given a set of maximal traces TTR, the corresponding set of heterogeneous traces $\psi(TTR)$ is defined as follows:*

- $TTR = \{\langle(\langle\surd\rangle, X)\rangle\}$.
  $\psi(TTR) \mathrel{\widehat{=}} \{\langle\surd\rangle\}$.
- $\exists ttr \in TTR \cdot \forall i : 1 \leq i \leq \#ttr \cdot \pi_1(ttr(i)) = \mathcal{A}^* \wedge \pi_2(ttr(i)) = \mathcal{A}$.
  $\psi(TTR) \mathrel{\widehat{=}} \mathcal{A}^*$.
- $\forall ttr \in TTR \cdot \forall i : 1 \leq i \leq \#ttr \cdot \pi_1(ttr(i)) = \langle\rangle \wedge \pi_2(ttr(i)) = \mathcal{A}$.
  $\psi(TTR) \mathrel{\widehat{=}} \{\langle\rangle\}$.
- $TTR = pref(ttr)$, *where* $ttr = \langle(\langle\rangle, \mathcal{A}), .., (\langle\rangle, \mathcal{A})\rangle$, *and* $\#ttr = n$.
  $\psi(TTR) \mathrel{\widehat{=}} pref(\langle set.n, reset\rangle)$.
- $pref(ttr) \subseteq TTR$, *where* $ttr = \langle(\langle\rangle, \mathcal{A}{-}C), .., (\langle\rangle, \mathcal{A}{-}C), (\langle\rangle, \mathcal{A}{-}C{-}B), (\langle\rangle, \mathcal{A}{-}B)\rangle$, $\#ttr = n{+}1$, *and B and C are finite sets of events,* $C = \{c_1, .., c_k\}$.
  Let $ttr_i = \langle(\langle\rangle, \mathcal{A}{-}C), .., (\langle\rangle, \mathcal{A}{-}C)\rangle \frown \langle(s_i, X_i)\rangle$, *where* $head(s_i) = c_i \in C$, *for* $i = 1, .., k$.
  $\psi(TTR) \mathrel{\widehat{=}} pref(\langle set.n, reset\rangle) \cup \{\langle set.n, reset\rangle \frown s \mid s \in \psi(TTR/ttr)\} \cup$
  $\bigcup_{i=1}^{k}(\ pref(\langle set.n, c_i, halt\rangle)$
  $\cup\{\langle set.n, c_i, halt\rangle \frown tail(s_i) \frown s \mid s \in \psi(TTR/ttr_i)\})$
- *For other cases,* $\psi(TTR) \mathrel{\widehat{=}} \cup\{flat(ttr) \mid ttr \in TTR\}$.

We assume that any sequential action can be written in the guarded normal form $\Box_{i=1}^{n}(c_i \to P_i)$. We construct Φ as follows.

Definition 4 (Syntactic Mapping).

$$
\begin{array}{rcl}
\Phi(Skip) & \mathrel{\widehat{=}} & Skip \\
\Phi(Chaos) & \mathrel{\widehat{=}} & Chaos \\
\Phi(Stop) & \mathrel{\widehat{=}} & Stop \\
\Phi(x := e) & \mathrel{\widehat{=}} & x := e \\
\Phi(b \& P) & \mathrel{\widehat{=}} & b \& \Phi(P) \\
\Phi(P \lhd b \rhd Q) & \mathrel{\widehat{=}} & \Phi(P) \lhd b \rhd \Phi(Q) \\
\Phi(Wait\ t) & \mathrel{\widehat{=}} & set!t \to reset? \to Skip \\
\Phi(\Box_{i=1}^{k}(c_i \to P_i)) & \mathrel{\widehat{=}} & \Box_{1 \leq i \leq k}(c_i \to \Phi(P_i)) \\
\Phi(P \Box Q) & \mathrel{\widehat{=}} & \Phi(P) \Box \Phi(Q) \\
\Phi(P \sqcap Q) & \mathrel{\widehat{=}} & \Phi(P) \sqcap \Phi(Q) \\
\Phi(P; Q) & \mathrel{\widehat{=}} & \Phi(P); \Phi(Q) \\
\Phi(P \backslash E) & \mathrel{\widehat{=}} & \Phi(P) \backslash E \\
\Phi((\Box_{i=1}^{n}(c_i \to P_i)) \overset{t}{\rhd} Q) & \mathrel{\widehat{=}} & set!t \to ((\Box_{i=1}^{n}(c_i \to halt! \to \Phi(P_i))) \Box (reset? \to \Phi(Q))) \\
\Phi(\mu X \bullet \Box_{i=1}^{n}(c_i \to P_i(X))) & \mathrel{\widehat{=}} & \mu X \bullet \Box_{i=1}^{n}(c_i \to \Phi(P_i(X)))
\end{array}
$$

Theorem 3. *The syntactic mapping Φ conforms with the semantic mapping ψ. That is, given any program P from* TCz*, we have*

$\psi([\![P]\!]_{time}) = [\![\Phi(P)]\!]$

The proof is straightforward by a structural induction on *P*.

Theorem 4. *The syntactic mapping Φ is a homomorphic solution to the equation (†).*

4.3 Algebraic Laws

The set of processes generated by function Φ are called as *heterogeneous communicating processes* (HCP). It can be regarded as an extension of Communicating Sequential Processes (CSP) (if we ignore state features). It enriched CSP with timer events, thus is also subject to the healthiness conditions for CSP (Chapter 8 of [8]). However, as timer events have the same behaviour in both synchronous and asynchronous models, it satisfies some additional healthiness conditions. These additional properties will yield a subset of CSP processes. Therefore, although heterogeneous communicating processes are an extension of CSP, they can be simulated by a subset of CSP.

We shall present the additional properties in what follows.

HC1 $[\![P]\!] \wedge utr^\frown \langle set_1, set_2\rangle \preceq utr' = [\![P]\!] \wedge utr^\frown \langle set_2, set_1\rangle \preceq utr'$

It states that, if a heterogeneous communicating process sets two timers consecutively, then it can set them in any order.

Similarly, we have the following healthiness conditions.

HC2 $[\![P]\!] \wedge utr^\frown \langle set_1, halt_2\rangle \preceq utr' = [\![P]\!] \wedge utr^\frown \langle halt_2, set_1\rangle \preceq utr'$

HC3 $[\![P]\!] \wedge utr^\frown \langle halt_1, halt_2\rangle \preceq utr' = [\![P]\!] \wedge utr^\frown \langle halt_2, halt_1\rangle \preceq utr'$

The following condition indicates that no heterogeneous communicating process can refuse both events *halt* and *reset* simultaneously when the timer is activated.

HC4 $[\![P]\!] \wedge utr' = utr_0^\frown \langle set\rangle^\frown utr_1 \wedge utr_1 \upharpoonright \{halt, reset\} = \langle\rangle \Rightarrow \{halt, reset\} \not\subseteq ref'$

In what follows, we give some expansion laws to transform a parallel action into a sequential one. Take note that timer events play different roles from normal events.

In the following laws, we assume P and Q are already in guarded normal forms: $P = \Box_{i=1}^{n}(c_i \to P_i)$, $Q = \Box_{k=1}^{m}(d_k \to Q_k)$, where for all i, k, $c_i \neq d_k$, and c_i, d_k are not timer events. Let $cs = (\alpha P \cap \alpha Q)\backslash\{set, halt, reset\}$.

The following one is the standard expansion law where no timer events are involved.

Law 1 $$P|[cs]|Q = \begin{cases} \Box_{i,k:c_i=d_k\in cs}(c_i \to (P_i|[cs]|Q_k)) \\ \Box\Box_{i:c_i\notin cs}(c_i \to (P_i|[cs]|Q)) \\ \Box\Box_{k:d_k\notin cs}(d_k \to (P|[cs]|Q_k)) \end{cases}$$

If timer events are involved, we should use the following expansion laws.

Law 2 $$(set_1 \to P)|[cs]|(set_2 \to Q) = (set_1 \to set_2 \to (P|[cs]|Q)) \sqcap (set_2 \to set_1 \to (P|[cs]|Q))$$

Note that the two output events set_1 and set_2 can occur in any order, which is reflected by the internal choice. So do the two *halt* events or a mix of them, as illustrated by the following two laws.

Law 3 $$(set_1 \to P)|[cs]|(halt_2 \to Q) = (set_1 \to halt_2 \to (P|[cs]|Q)) \sqcap (halt_2 \to set_1 \to (P|[cs]|Q))$$

Law 4 $(halt_1 \rightarrow P)|[cs]|(halt_2 \rightarrow Q) = (halt_1 \rightarrow halt_2 \rightarrow (P|[cs]|Q)) \sqcap (halt_2 \rightarrow halt_1 \rightarrow (P|[cs]|Q))$

The events *set* and *halt* from the master process have higher priority than the *reset* event emitted by the slave process. This is reflected in the following two laws:

Law 5 $(set_1 \rightarrow P)|[cs]|(reset_2 \rightarrow Q) = set_1 \rightarrow (P|[cs]|(reset_2 \rightarrow Q))$

Law 6 $(halt_1 \rightarrow P)|[cs]|(reset_2 \rightarrow Q) = halt_1 \rightarrow (P|[cs]|(reset_2 \rightarrow Q))$

Law 7 $(reset_1 \rightarrow P)|[cs]|(reset_2 \rightarrow Q) = (reset_1 \rightarrow (P|[cs]|(reset_2 \rightarrow Q))) \Box (reset_2 \rightarrow ((reset_1 \rightarrow P)|[cs]|Q))$

Take note that different from **Law 2**, external choice is used here as the two input events $reset_1$ and $reset_2$ have to wait for the corresponding output events from the environment (Timer processes).

5 Related Work and Conclusion

The two mostly related integrated formal specification languages are TCOZ [11] and *Circus* [21]. *Circus* is a combination of CSP and Z. It also includes specification statements found in Morgan's refinement calculus [13] and Dijkstra's language of guarded commands [3]. *Circus* has a well-defined syntax and a formal semantics [23, 21] based on Hoare and He's unifying theories of programming [8]. Case studies using the language are explored in [22] to show its power of expressiveness. A development method for *Circus* using refinement is described in [15]. A timed model for *Circus* was provided in [17]. Our untimed model CZ is a subset of *Circus*.

TCOZ is a blending of Object-Z [5, 18] and Timed CSP [16, 2], aiming at specification for complex real-time systems. The semantic link between the two formalisms Timed CSP and Object-Z is reported in [12]. TCOZ was enriched with sensors/actuators in [10]. A unified observation model for TCOZ is presented in [14]. Recent work [4] proposed a projection from TCOZ specifications to Timed Automata Patterns for model-checking timing properties using UPPAAL. Their syntactical mapping is proved sound under bisimulation. In our paper, we propose to verify timing properties in untimed framework by constructing a property-oriented untimed model, which, we believe, should be much simpler than doing it within the timed model.

Instead of using the same complex model as both semantic and reasoning models, we advocate the construction of small property-oriented models, that are separated from the whole semantic model, for verification of particular kinds of properties. In our instantiation in terms of timing properties, the approach does make analysis and reasoning about certain timing properties simpler and easier. A deep semantic link has been built between the timed model and the untimed model in the observation level, which ensures that it is safe to use a smaller property-oriented model for verification.

References

1. M. Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Aspects of computing*, 12:182–196, 2000.
2. J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.
3. E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453 – 457, 1975.
4. J. S. Dong, P. Hao, S.C. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In *6th International Conference on Formal Engineering Methods, ICFEM 2004*, Seattle, WA, USA, November 2004.
5. R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing Series. Macmillan, March 2000.
6. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.
7. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
8. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
9. L. Li and J. He. Towards a Denotational Semantics of Timed RSL using Duration Calculus. Technical Report 161, UNU/IIST, April 1999.
10. B. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. In *FM'99: World Congress on Formal Methods*, volume 1709 of *Lect. Notes in Comput. Sci.*, Toulouse, France, September 1999. Springer-Verlag.
11. B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.
12. B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.
13. C. C. Morgan. *Programming from Specifications*. Prentice Hall, 1994.
14. S.C. Qin, J.S. Dong, and W.N. Chin. A Semantics Foundation for TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lect. Notes in Comput. Sci.*, pages 321–340. Springer, 2003.
15. A. Sampaio, J. Woodcock, and A. Cavalcanti. Refinement in Circus. In *FME2002: International Symposium of Formal Methods Europe*, volume 2391 of *Lect. Notes in Comput. Sci.*, pages 451–470. Springer-Verlag, 2002.
16. S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lect. Notes in Comput. Sci.*, pages 640–675. Springer-Verlag, 1992.
17. A. Sherif and J. He. Towards a Timed Model for Circus. In C. George and H. Miao, editors, *ICFEM'02 Formal Methods and Software Engineering*, volume 2495 of *Lect. Notes in Comput. Sci.*, pages 613–624. Springer-Verlag, 2002.
18. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
19. G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In *International Conference on Formal Engineering Methods*, pages 293–302. IEEE Computer Society, 1997.
20. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, Prentice-Hall, 1992.

21. J. Woodcock and A. Cavalcanti. Circus: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wofson Building, Parks Road, Oxford OX1 3QD, UK, July 2001.
22. J. Woodcock and A. Cavalcanti. The steam boiler in a unified theory of Z and CSP. In J. He, Y. Li, and G. Lowe, editors, *The 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 291–298. IEEE Press, 2001.
23. J. Woodcock and A. Cavalcanti. The Semantics of Circus. In *2nd International Conference on Z and B*, volume 2272 of *Lect. Notes in Comput. Sci.*, pages 184–203. Springer-Verlag, 2002.